บทที่ 3

ลิสต์ สแตก และคิว

บทนี้เราจะมาเรียนโครงสร้างข้อมูลพื้นฐานอีกจำนวนหนึ่ง นั่นก็คือ ลิสต์ (list) สแตก (stack) และคิว (queue) หรือแถวคอย

ลิสต์

โครงสร้างของลิสต์นั้น จริงๆก็คือ มีอะไรมาเรียงกันนั่นเอง ดังนั้นอาร์เรย์ก็ถือเป็นรูปแบบหนึ่งของลิสต์ได้ เริ่มแรกเราควรจะมาดูว่า เราจะทำอะไรกับลิสต์ได้บ้าง ตาราง 3.1 แสดงสิ่งที่เราควรจะสามารถทำกับลิสต์ได้

ตาราง 3.1 ฟังก์ชั่นของลิสต์และชื่อภาษาอังกฤษที่เป็นที่รู้จักกันทั่วไปของแต่ละฟังก์ชั่น

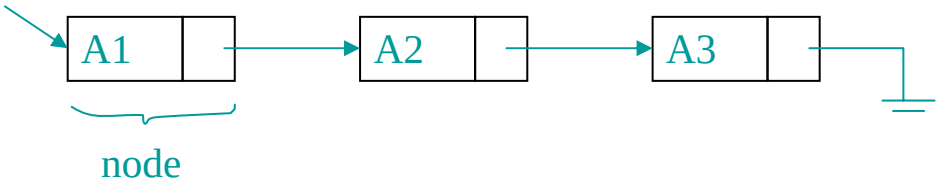
find	หาตำแหน่งที่อยู่ของสมาชิกตัวหนึ่ง
insert	ใส่สมาชิกใหม่ลงในตำแหน่งที่กำหนด
findKth	รีเทิร์นสมาชิกตัวที่ k
remove	เอาสมาชิกที่กำหนดออกจากลิสต์
head	รีเทิร์นสมาชิกตัวแรกในลิสต์
tail	รีเทิร์นลิสต์ที่เอาสมาชิกตัวแรกออกไปแล้ว
append	เอาลิสต์สองลิสต์มาต่อกัน

ลิงค์ลิสต์ (Linked list)

อย่างที่เขียนไว้ในตอนแรกว่า อาร์เรย์ก็ถือเป็นลิสต์แบบหนึ่ง ถ้าเราใช้อาร์เรย์เป็นลิสต์ จะเกิดอะไรขึ้น อย่างแรกก็คือ ลิสต์ที่ทำด้วยอาร์เรย์นั้นมีจำนวนสมาชิกจำกัด เพราะว่าเราต้องกำหนดขนาดของอาร์เรย์ก่อนใช้งาน

ฟังก์ชั่น find จะต้องใช้เวลา O(n) เพราะว่า find นั้นต้องหาสมาชิกที่เราต้องการด้วยการหาแบบเรียงตัว นับจากตัวแรกของอาร์เรย์ ฟังก์ชั่น findKth(i) นั้นจะใช้เวลาคงที่ เพราะใช้ดัชนีของอาร์เรย์หาสมาชิกตัวที่เราต้องการได้ทันที ส่วนฟังก์ชั่น insert กับ remove จะใช้เวลานานเพราะอาจต้องเลื่อนทุกๆสมาชิกในอาร์เรย์เมื่อมีการเติมหรือเอาสมาชิกตัวใดตัวหนึ่งออกจากอาร์เรย์ (ถ้าเราใส่สมาชิกไปที่หัวอาร์เรย์ ส่วนอื่นๆในช่องถัดไปก็ต้องเลื่อนช่องหมด)

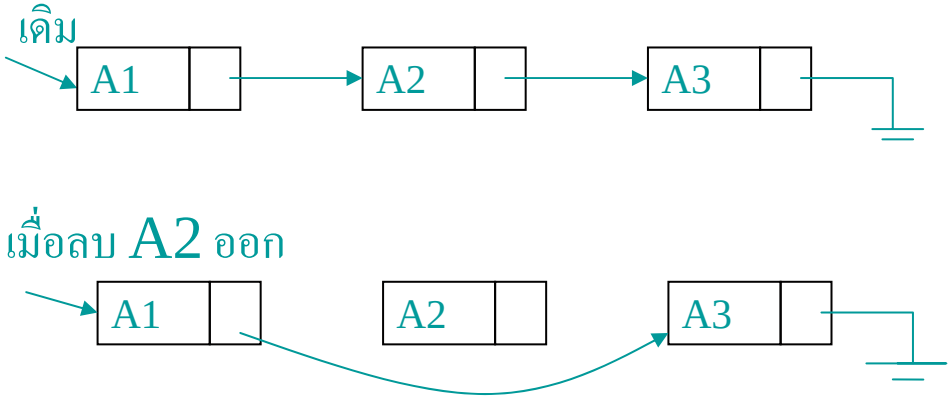
ดังนั้นการใช้อาร์เรย์ทำลิสต์จึงไม่เหมาะเสียทีเดียว ภาษาเชิงวัตถุอย่างจาวาได้ก่อให้เกิดแนวคิดในการสร้างลิสต์อีกวิธีการหนึ่ง เรียกว่า ลิงค์ลิสต์ (linked list) ลักษณะของลิงค์ลิสต์ก็คือ มีวัตถุ (เรียกอีกอย่างว่าโนด - node) ที่เก็บค่าหรือวัตถุตัวอื่นไว้ แล้วมี reference ต่อกันเป็นทอดๆ ตัวสุดท้ายจะไม่มี reference ไปถึงอะไร (เรียกอีกอย่างว่า มี reference เป็น null) รูปที่ 3.1 แสดงลักษณะของลิงค์ลิสต์



รูป 3.1 ลักษณะของลิงค์ลิสต์

ในรูปแบบใหม่นี้ การ find ใช้เวลา O(n) เหมือนเดิม เพราะยังต้องหาเรียงตัว นับจากตัวแรก ส่วน findKth(i) นั้นใช้เวลา O(i) ซึ่งมากกว่าตอนที่เราใช้อาร์เรย์เพราะต้องหาเรียงตัว

แต่ส่วนที่ดีของการใช้ลิงค์คือจะประหยัดเวลาในการ insert และ remove การลบของออกจากลิสต์ทำได้ง่ายขึ้นเพราะแค่เอา reference ข้ามตัวที่ต้องการลบไปก็พอแล้ว ดังรูปที่ 3.2



รูป 3.2 การเอาสมาชิก A2 ออกจากลิงค์ลิสต์

การเปลี่ยนพอยต์เตอร์ (หรือ reference) เพื่อลบ A2 ออก ทำได้ดังนี้

- เก็บตัวพอยต์เตอร์ ที่ชี้จาก A2 ไปเอาไว้ในตัวแปร แล้วให้ A2 ชี้ไปที่ null (จริงๆไม่ต้องให้ A2 ชี้ไปที่ null ก็ได้เพราะพอยต์เตอร์ นี้ก็ไม่ทำให้เข้าถึง A2 ได้อยู่ดี)
- เอาพอยต์เตอร์ ที่ชี้ไปที่ A2 เปลี่ยนไปให้ชี้ตัวแปรที่เก็บไว้

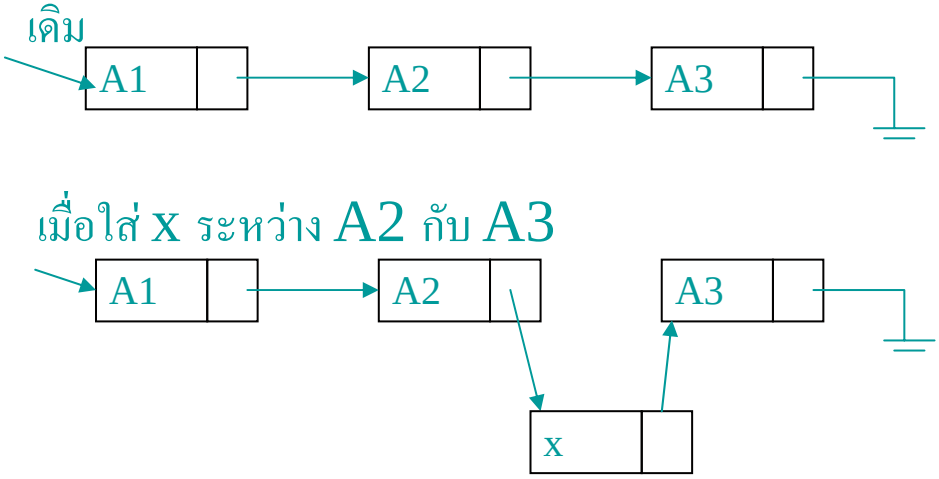
ถ้าเอา A2 ออกแล้ว เมื่อไม่มี reference เข้าถึง A2 ได้อีก จาวาจะลบ A2 ไปเองเพราะจาวามีกระบวนการ garbage collection ซึ่งจัดการกับส่วนความจำที่ใช้เก็บวัตถุต่างๆได้เอง

การ insert x ก็จะใช้การเปลี่ยนพอยต์เตอร์ (หรือ reference) เช่นเดียวกัน รูป 3.3 แสดงการเปลี่ยน reference สำหรับการ insert โดยเอาพอยต์เตอร์ จาก A2 ชี้ไปที่ x และเอา พอยต์เตอร์ จาก x ชี้ไปที่ A3

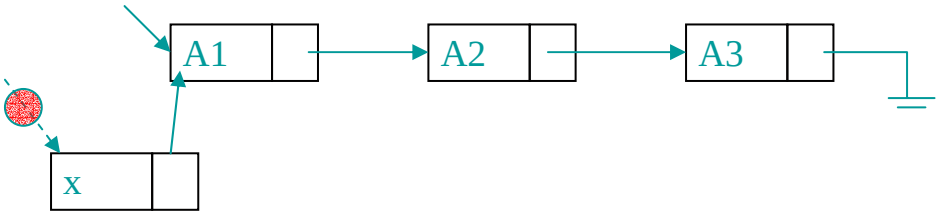
ถ้าเราทำการ insert ของลงไปข้างหน้าโนดแรก หลักการยังเหมือนเดิม ต่างกันตรงรายละเอียด รูปที่ 3.4 แสดงการ insert ที่หัวลิสต์ เนื่องจากไม่มีโนดที่อยู่ก่อน A1 ดังนั้นจึงตั้งค่าพอยต์เตอร์

จากโนดนั้นไม่ได้ แต่เราสามารถตั้งค่าพอยต์เตอร์จาก x มาได้ แล้วจึงตั้งค่าพอยต์เตอร์จากหัวลิสต์ A1 มาที่ x เพื่อให้เป็นหัวลิสต์ใหม่

การเอาของออกจากหัวลิสต์ก็จะทำให้ต้องโค้ดในกรณีพิเศษเหมือนการเอาของใส่ที่หัวลิสต์

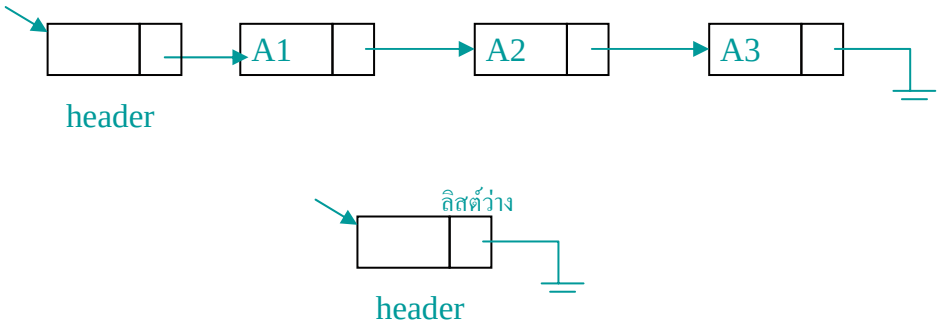


รูป 3.3 การใส่ x ลงในลิงค์ลิสต์



รูป 3.4 การเติมของที่หัวลิสต์

เพื่อหลีกเลี่ยงการโค้ดด้วยกรณีพิเศษสำหรับการใส่สมาชิกลงหัวลิสต์หรือการนำสมาชิกออกจากหัวลิสต์ เราสามารถให้มีหัวลิสต์ปลอม เรียกว่า header หรือ dummy node ได้ ถ้าทำอย่างนี้ทุกๆ โนดก็จะมีโนดนำหน้า ทำให้รายละเอียดการโค้ดเหมือนกันหมด รูป 3.5 แสดงลิงค์ลิสต์ซึ่งมีหัวลิสต์ปลอม โดยลิสต์ว่างนั้นจะเป็นหัวลิสต์ปลอมซึ่งอยู่เดี่ยวๆ



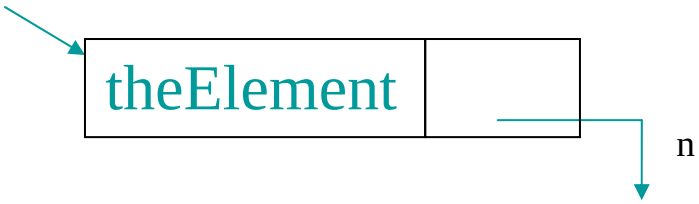
รูป 3.5 ลิงค์ลิสต์ซึ่งใช้หัวลิสต์ปลอม

ต่อไปเรามาดูโค้ดกัน ในที่นี้เราจะแบ่งโค้ดออกเป็นสามส่วน คือส่วนโนด ส่วนการชี้โนดที่สนใจ และส่วนลิสต์ ก่อนอื่นเรามาดูที่ส่วนโนดกัน โค้ดในรูปที่ 3.6 แสดงโค้ดของโนด โค้ดสำหรับลิงค์ลิสต์ในบทนี้ใช้ต้นแบบจากหนังสือของ Mark Allen Weiss[1]

```
1:      class ListNode{
2:            Object   element;
3:            ListNode next;
4:            // Constructors
5:            ListNode( Object theElement )
6:            {
7:                this( theElement, null );
8:            }
9:
10:           ListNode( Object theElement, ListNode n )
11:           {
12:               element = theElement;
13:               next    = n;
14:           }
15:       }
```

รูป 3.6 โค้ดของโนด

โนดในรูปที่ 3.6 นี้มีคอนสตรัคเตอร์ตัวที่สองเป็นหลัก ตัวแรกเมื่อถูกเรียกใช้ก็จะไปเรียกตัวที่สองอีกต่อหนึ่ง เพียงแค่ให้ตัวที่ชี้ไปโนดถัดไปชี้ไปที่ null ส่วนตัว n ในบรรทัดที่สิบคือโนดอีกตัวนั่นเอง ดังนั้นโนดที่เราสร้างขึ้นในที่นี้จึงมีลักษณะดังรูปที่ 3.7



รูป 3.7 โนดที่สร้างจากโค้ดในรูป 3.6

ต่อไปจะกล่าวถึงโค้ดในส่วนการชี้โนดที่สนใจ ส่วนนี้เราจะเรียกว่า อิเทอเรเตอร์ (iterator) ซึ่งเป็นแนวคิดที่ได้ถูกนำมาใช้ในโครงสร้างข้อมูลหลายๆแบบในปัจจุบัน ในจาวาเองก็มีโครงสร้างข้อมูลที่ให้มาโดยมีอิเทอเรเตอร์ประกอบ ซึ่งจะซับซ้อนกว่าตัวอย่างที่จะแสดงต่อไปนี้ แต่ก็สร้างด้วยหลักการเดียวกัน อิเทอเรเตอร์ของเราอยู่ในรูปที่ 3.8

ลิสต์อิเทอเรเตอร์ คือ object ที่ชี้ไปยังโนดที่เราสนใจในลิสต์ บางคนอาจสงสัยว่าทำไมต้องเขียนคลาสนี้แยกจากลิสต์ในเมื่อเราก็เก็บตำแหน่งที่สนใจไว้ในลิสต์ได้เอง นั่นก็เพราะว่าเราจะได้เก็บตำแหน่งที่สนใจได้หลายตำแหน่ง ถ้าเราเก็บบนลิสต์เลย ก็คงต้องมีการจำกัดจำนวนตำแหน่งที่เก็บได้ แต่เมื่อเราแยกเก็บ ก็ทำให้มีกี่ตำแหน่งที่สนใจก็ได้

จากโค้ดในรูป 3.8 จะเห็นได้ว่าอิเทอเรเตอร์หนึ่งตัวเก็บพอยต์เตอร์ไปยังโนดหนึ่งตัว ในคอนสตรัคเตอร์ของอิเทอเรเตอร์นั้นเราสร้างอิเทอเรเตอร์ขึ้นมาให้ชี้ไปยังโนดที่เราสนใจทันที

คราวนี้มาดูตัวลิสต์เลย ซึ่งโค้ดนั้นอยู่ในรูปที่ 3.9 ในโค้ดนี้เราใช้หัวลิสต์ปลอม (dummy node) จากโค้ด ตอนที่ทำการ insert เราต้องเช็คว่า p มีตัวตน และชี้ไปที่โนดหนึ่งจริงหรือเปล่า ไม่งั้นจะทำไม่ได้ การเปลี่ยน pointer เพื่อการ insert จะเหมือนที่แสดงในรูป3.3 เพียงแต่เราเข้าถึงโนดที่อยู่ก่อนตัวที่จะเติมลงไป ได้โดยใช้อิเทอเรเตอร์

การ find นั้นเป็นการวนลูปเรื่อยๆ โดยเริ่มจากโนดแรกที่มีของ จนกว่าจะเจอ x หรือสิ้นสุดลิสต์ แล้วจึงสร้างอิเทอเรเตอร์ที่ชี้ไปยังที่ที่ x อยู่ (หรือชี้ไปที่ null ถ้า x ไม่อยู่ในลิสต์เลย)

ส่วน findPrevious นั้นการวนลูปจะเกือบเหมือนกับ find เพียงแต่ว่า ใช้ itr.next แทน itr เพื่อให้ตรวจสอบได้ยังโนดถัดไป เมื่อเจอ x ในโนดถัดไป จะได้ รีเทิร์นอิเทอเรเตอร์ที่ชี้ไปโนดปัจจุบัน

(ซึ่งอยู่ก่อน x พอดี) ส่วนถ้าไม่เจอ x เราก็ยังได้รู้ว่าโนดถัดไปจะเป็น null เราจะได้รีเทิร์นอิเทอเรเตอร์ที่ชี้ไปโนดปัจจุบันซึ่งเป็นโนดสุดท้ายพอดี

```
1:    public class LinkedListItr{
2:          ListNode current;    // ตำแหน่งปัจจุบันที่เราสนใจ
3:          LinkedListItr( ListNode theNode )
4:          {
5:              current = theNode;
6:          }
7:
8:          /**
9:           * ดูว่า current เลยท้ายลิสต์ไปหรือยัง
10:          * @return true ถ้า current เป็น null
11:          */
12:         public boolean isPastEnd( )
13:         {
14:             return current == null;
15:         }
16:
17:         /**
18:          * @return item ที่เก็บไว้ใน current หรือไม่ก็ null ถ้าตำแหน่งของ
19:          * current ไม่ได้อยู่ในลิสต์
20:          */
21:         public Object retrieve( )
22:         {
23:             return isPastEnd( ) ? null : current.element;
24:         }
25:
26:         /**
27:          * เขยิบ current ไปยังตำแหน่งถัดไปในลิสต์ ถ้า current เป็น null ก็ไม่ต้อง
28:          * ทำอะไร
29:          */
30:         public void advance( )
31:         {
32:             if( !isPastEnd( ) )
33:                 current = current.next;
34:         }
35:     }
```

รูป 3.8 โค้ดของลิสต์อิเทอเรเตอร์

ส่วน printList นั้นเป็นการพิมพ์เนื้อในของแต่ละโนดเรียงไปจนหมด โดยใช้การลูปที่อิเทอเรเตอร์ตั้งแต่สมาชิกตัวแรกของลิสต์จนถึงสมาชิกตัวสุดท้าย ใช้เมธอดของอิเทอเรเตอร์ทั้งหมด

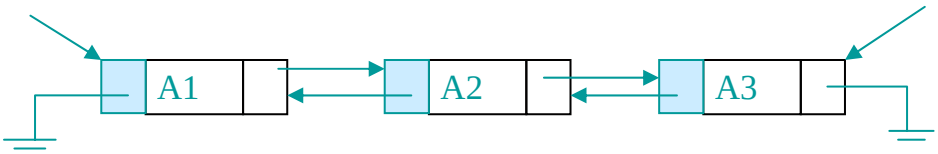
```
1:    public class LinkedList{
2:        ListNode header; //เข้าได้จากหัวลิสต์
3:        public LinkedList( ){
4:                header = new ListNode( null );
5:        }
6:
7:        public boolean isEmpty( ){
8:            return header.next == null;
9:        }
10:
11:       public void makeEmpty( ){
12:           header.next = null;
13:       }
14:
15:       /**
16:        * รีเทิร์น iterator ที่ชี้ไป header node.
17:        */
18:       public LinkedListItr zeroth( ){
19:           return new LinkedListItr( header );
20:       }
21:
22:       /**
23:        * รีเทิร์น iterator ที่ชี้ไป node ถัดจาก header (ซึ่งเป็น null ได้
24:        * ถ้าลิสต์นี้ว่าง)
25:        */
26:       public LinkedListItr first( ){
27:           return new LinkedListItr( header.next );
28:       }
29:
30:       /**
31:        * ใส่โนดใหม่ตามหลังสมาชิกที่ชี้ด้วย p
32:        * @param x item ที่จะเอาใส่โนดใหม่
33:        * @param p เป็น iterator ชี้ตำแหน่งที่อยู่ก่อนโนดที่จะลงใหม่
34:        */
35:       public void insert(Object x,LinkedListItr p){
36:           if( p != null && p.current != null )
37:               p.current.next = new ListNode( x,
38:                                       p.current.next );
39:       }
40:
41:       /**
42:        * @param x คือ ของข้างในของโนดที่เราต้องการหา.
43:        * @return iterator ที่ชี้ไปที่โนดแรกที่มี x อยู่ข้างใน หรือชี้ไปที่ null ถ้า x
44:        *ไม่อยู่ ในลิสต์เลย
45:       /*
46:       public LinkedListItr find(Object x){
```

```
47:          ListNode itr = header.next;
48:          while( itr!=null &&!itr.element.equals(x))
49:             itr = itr.next;
50:          return new LinkedListItr( itr );
51:       }
52:
53:       /**
54:       * รีเทิร์น iterator ที่ชี้ไปที่โนดก่อนโนดแรกที่มี x
55:       * ถ้าไม่มี x ในลิสต์เลยให้รีเทิร์น iterator ที่ชี้ไปที่โนดสุดท้ายของลิสต์
56:       */
57:       public LinkedListItr findPrevious(Object x){
58:          ListNode itr = header;
59:          while( itr.next!=null
60:                   &&!itr.next.element.equals( x ) )
61:             itr = itr.next;
62:          return new LinkedListItr( itr );
63:       }
64:
65:       /**
66:        * เอาโนดของ x ตัวแรกที่เจอออกจากลิสต์
67:        * @param x คือ item ในโนดที่ต้องการเอาออก
68:        */
69:        public void remove(Object x){
70:            LinkedListItr p = findPrevious( x );
71:            if( p.current.next != null )
72:            // นี่หมายความว่าหา x เจอ เพราะไม่ใช่ตัวสุดท้ายของลิสต์
73:                //เปลี่ยน reference ข้ามตัวที่มี x ไป
74:                p.current.next = p.current.next.next;
75:        }
76:
77:        public static void printList(LinkedList theList)
78:        {
79:            if(theList.isEmpty( ) )
80:              System.out.print( "Empty list" );
81:            else
82:            {
83:              LinkedListItr itr = theList.first( );
84:              for(;!itr.isPastEnd( );itr.advance( ) )
85:                System.out.print(itr.retrieve( )+" " );
86:            }
87:              System.out.println( );
88:        }
89:   }
```

รูป 3.9 โค้ดของลิงค์ลิสต์

ลิงค์ลิสต์สองทาง(Doubly-linked list)

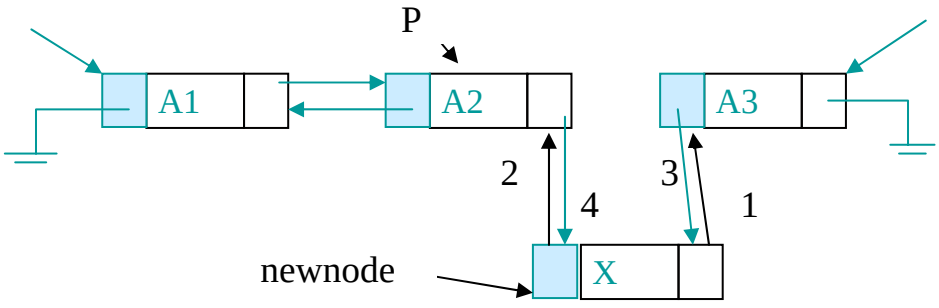
เป็นลิงค์ลิสต์แบบพิเศษ คือ ในโนดจะมีตัวแปรเพิ่มมาอีกหนึ่งตัว คือ previous ซึ่งจะชี้ไปยังโนดที่อยู่ก่อนหน้า ทำหน้าที่คล้าย next เพียงแต่ชี้ไปคนละทางเท่านั้น การมีลิสต์แบบนี้ทำให้เราสามารถเรียงดูลิสต์ได้สองทิศทาง แต่ก็มีข้อเสียคือ ต้องใช้เวลาเพิ่มในการเปลี่ยนพอยต์เตอร์ รูป 3.10 แสดงลักษณะของลิงค์ลิสต์สองทาง



รูป 3.10 ลิงค์ลิสต์สองทาง

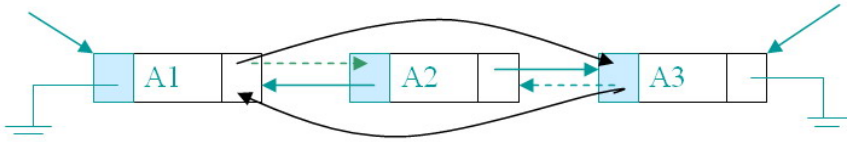
ในการจะใส่โนดใหม่ลงไปในลิสต์แบบนี้นั้น ลำดับในการเปลี่ยนพอยต์เตอร์มีความสำคัญมาก ถ้าเราใช้ลำดับไม่ถูกต้อง จะทำให้ไม่สามารถเปลี่ยนพอยต์เตอร์ได้หมด สมมุติว่าเรากำลังจะแทรกโนดใหม่ (มีพอยต์เตอร์ไปถึง เรียกว่า newnode) เข้าระหว่างโนด A2 และ A3 ในรูป 3.10 (เรามีพอยต์เตอร์ p ชี้ไปที่ A2 อยู่แล้ว เพื่อความเข้าใจที่ง่ายเราจะไม่ใช้อิเทอเรเตอร์) เราอาจทำการเปลี่ยนพอยต์เตอร์ตามลำดับดังนี้ (ให้พอยต์เตอร์ที่ชี้ไปด้านขวาเรียกว่า next ส่วนพอยต์เตอร์ที่ชี้ไปด้านซ้ายเรียกว่า previous) ลำดับการเปลี่ยนจะเป็นดังรูป 3.11

- newnode.next = p.next;
- newnode.previous = p.next.previous;
- p.next.previous = newnode;
- p.next = newnode;



รูป 3.11 การเติมโนดใหม่ลงในลิงค์ลิสต์สองทาง

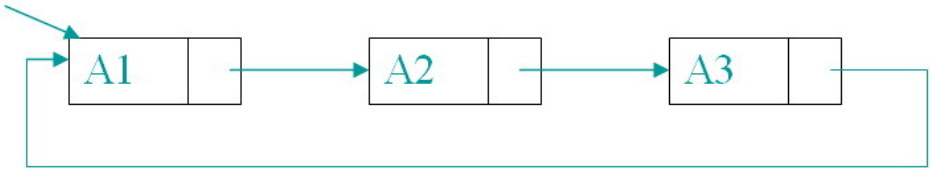
ส่วนการลบสมาชิกออกจากลิงค์ลิสต์แบบสองทางนั้นไม่ต่างจากการลบสมาชิกออกจากลิงค์ลิสต์แบบธรรมดามากนัก แค่ทำการเปลี่ยนลิงค์ให้ข้ามไปแบบในรูปที่ 3.12



รูป 3.12 การเอาสมาชิกออกจากลิงค์ลิสต์แบบสองทาง

ลิงค์ลิสต์แบบต่อเป็นวง(Circular linked list)

ลิงค์ลิสต์แบบนี้มีลักษณะเหมือนลิงค์ลิสต์ธรรมดา แต่ตัวท้ายสุดจะลิงค์กลับไปตัวแรก ดังรูปที่ 3.13



รูป 3.13 ลิงค์ลิสต์แบบต่อเป็นวง

การทำลิสต์แบบนี้ทำให้ไม่ต้องมีหัวลิสต์ปลอมก็ได้ เราเอามาทำให้เป็นลิงค์ลิสต์สองทางด้วยก็ยังได้ แล้วแต่จะประยุกต์

ต่อไปผมจะกล่าวถึงตัวอย่างการใช้งานของลิงค์ลิสต์ในรูปแบบต่างๆซึ่งซับซ้อนและต้องการการประยุกต์มากกว่าลิงค์ลิสต์แบบสองทางและแบบต่อเป็นวง

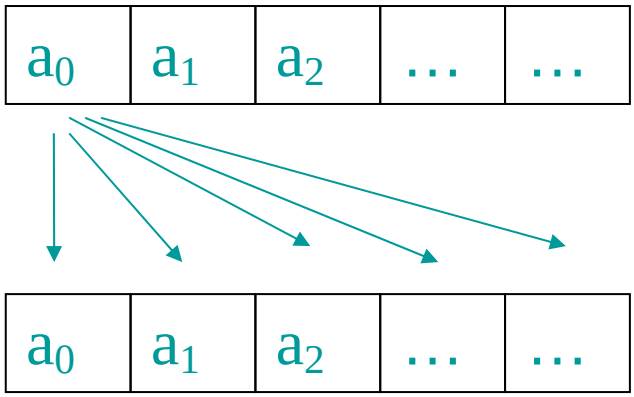
การใช้ลิงค์ลิสต์ลดพื้นที่ข้อมูล

ถ้าเก็บข้อมูลบางชนิดในอาร์เรย์ เราอาจต้องใช้พื้นที่มากในการเก็บ การใช้ลิงค์ลิสต์เก็บข้อมูลนั้นแทนอาจทำให้ลดพื้นที่ที่ต้องใช้ในหน่วยความจำได้เป็นจำนวนมาก ดังตัวอย่างต่อไปนี้

ตัวอย่างที่ 3-1
สมมติเราต้องการเก็บข้อมูลในรูปแบบของโพลิโนเมียล (polynomial) $\sum_{i=0}^{n} a_i x^i$

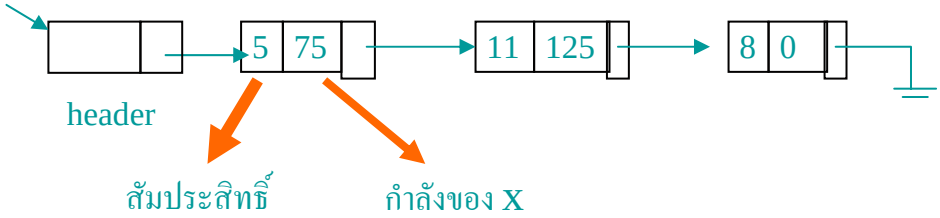
จริงๆ เราอาจใช้อาร์เรย์ โดยให้ตำแหน่งที่มีดัชนี i ใช้เก็บสัมประสิทธิ์ของ x^i ฉะนั้นการเอาโพลิโนเมียลสองจำนวนบวกกัน คำตอบจะเกิดจากการบวกอาร์เรย์ช่องต่อช่อง เพราะแต่ละช่องมีค่ากำลังของ x เท่ากัน

แต่ปัญหาจริงๆนั้นเกิดที่การคูณ เมื่อเอาสองโพลิโนเมียลคูณกันจะต้องเอาข้างในของแต่ละช่องมาคูณกับทุกช่องของpolynomial อีกตัว แล้วจึงเอาผลมาบวกกัน รูป 3.14 แสดงการคูณของ a0 ในอาร์เรย์แรก กับทุกๆเทอมในอาร์เรย์ที่สอง ดังนั้น ถ้ามีเทอมที่มีสัมประสิทธิ์เป็น 0 อยู่เป็นจำนวนมากล่ะก็ จะเป็นการคูณ 0 ไปโดยเสียเวลาเปล่า



รูป 3.14 การใช้อาร์เรย์จัดการผลคูณของโพลิโนเมียล

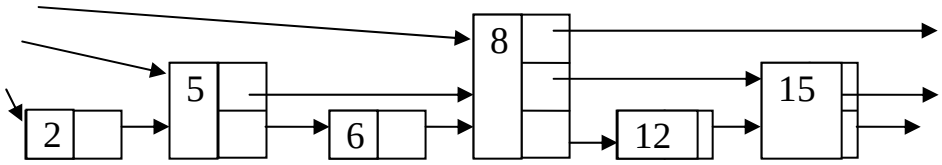
จากปัญหาของการใช้งานอาร์เรย์นี้ ทำให้เกิดแนวคิดที่จะใช้ลิงค์ลิสต์เก็บข้อมูลของโพลิโนเมียล รูปที่ 3.15 แสดงโครงสร้างการเก็บข้อมูล $5x^{75}+11x^{125}+8$ จะเห็นว่า ลดการใช้ 0 มากๆ ได้และใช้เนื้อที่ในการเก็บข้อมูลน้อยลง การบวกอาจต้องเสียเวลาในการหาเทอมที่มีค่ากำลังของ x เท่ากัน แต่การคูณนั้นจะง่ายขึ้นอย่างเห็นได้ชัด



รูป 3.15 การเก็บข้อมูลโพลิโนเมียลโดยใช้ลิงค์ลิสต์

ลิสต์แบบลิงค์ข้าม(Skip List)

จุดอ่อนของลิงค์ลิสต์ก็คือ ในเวลาจะหาของในลิสต์ต้องหาเรียงจากสมาชิกจากหัวลิสต์ไปท้ายลิสต์ วิธีหนึ่งที่จะช่วยแก้ปัญหานี้คือใช้ลิสต์แบบลิงค์ข้าม นั่นคือ ในโนดหนึ่ง เราให้มีพอยต์เตอร์แบบ next มากกว่าหนึ่งพอยต์เตอร์ได้ ซึ่งพอยต์เตอร์ที่มีเพิ่มมานั้นใช้ชี้ข้ามไปยังส่วนต่างๆ ของลิสต์



รูป 3.16 ลิงค์ลิสต์แบบลิงค์ข้าม

ลิงค์ในรูป 3.16 เป็นไปตามนิยามว่า โนดทุกลำดับชี้ไปยังโนดถัดไป โนดลำดับที่หารสองลงตัวชี้ไปยังลำดับที่หารสองลงตัวตัวถัดไป และโนดลำดับที่หารสี่ลงตัวชี้ไปยังโนดลำดับที่หารสี่ลงตัวตัวถัดไป

ลิงค์แบบข้ามนี้มีประโยชน์ในการหาของในลิสต์ที่มีของเรียงไว้อยู่แล้ว แต่ไม่ทราบว่าของแต่ละชิ้นอยู่ที่ไหน วิธีการหาของในลิสต์แบบนี้เริ่มจากการหาด้วยลิงค์ระดับสูงสุด (กระโดดข้ามมากที่สุด) ก่อน ถ้าไล่ตามลิงค์ระดับสูงสุดหาจนหมดลิสต์แล้วยังไม่เจอ ก็เริ่มหาใหม่โดยใช้ลิงค์ระดับที่ต่ำลงมา ทำเช่นนี้ไปเรื่อยๆ ถ้าระหว่างที่หา เกิดเจอโนดที่ต้องอยู่ข้างหลังโนดที่เราต้องการแน่ๆ ก็ให้ย้อนกลับมาหนึ่งโนด แล้วเริ่มการค้นหาจากโนดนั้นโดยใช้ลิงค์ระดับที่ต่ำลง

มา จากรูป ถ้าเราต้องการหา 15 เริ่มแรกใช้ลิงค์ระดับสูงมาดูที่ 8 แล้วดูตัวถัดไป แต่ว่ามันเกิน 15 ดังนั้นเราจึงเริ่มที่ 8 แล้วหาด้วยลิงค์ระดับต่ำลงมา จึงเจอ 15 จะเห็นว่าใช้การตามลิงค์เป็นจำนวนน้อยครั้งกว่าการตามลิงค์ตั้งแต่สมาชิกตัวแรก

ในกรณีที่หาของเจอ เวลาเฉลี่ยนั้นเป็น O(log n) ส่วนกรณีที่แย่ที่สุดนั้นเกิดจากการหาไม่เจอซึ่งต้องทำให้เริ่มหาใหม่จากลิงค์ระดับต่ำสุด ซึ่งกินเวลา O(n)

แม้ว่าการใช้ลิสต์แบบลิงค์ข้ามจะสามารถช่วยแก้ปัญหาการค้นหาข้อมูลได้ แต่ก็ยังมีปัญหาอื่นเกิดขึ้นตามมา ถ้าเราต้องการรักษาโครงสร้างของลิสต์ให้ลิงค์กันในแต่ละระดับเป็นช่องว่างสม่ำเสมอ การใส่สิ่งของชิ้นใหม่ลงไปในลิสต์หรือการลบของชิ้นหนึ่งออกจากลิสต์จะทำให้เราต้องเปลี่ยนโครงสร้างลิงค์ของแต่ละโนดใหม่ทั้งหมด ซึ่งเป็นความยุ่งยาก ฉะนั้นในทางปฏิบัติจึงบังคับแค่จำนวนของลิงค์ในแต่ละระดับ

ถ้าเราจะให้มีลิงค์ 4 ระดับ โดยระดับ n เริ่มจากโนดลำดับที่ 2^n และลิงค์ข้ามไปเป็นจำนวน 2^n โนด สมมติว่ามีจำนวนโนดตอนเริ่มต้นอยู่ 20 โนด จะต้องมีโนดที่มีลิงค์ระดับต่างๆดังนี้

- ระดับที่ 0 –> 20 โนด เพราะระดับล่างสุดต้องเป็นลิงค์ลิสต์แบบปกติ
- ระดับที่ 1 – >10 โนด
- ระดับที่ 2 –> 5 โนด
- ระดับที่ 3 –> 2 โนด

ถ้าเราถือว่าไม่นับโนดซ้ำ จะมีโนด

- ระดับที่ 3 –> 2 โนด
- ระดับที่ 2 –> 5-2 = 3 โนด
- ระดับที่ 1 -> 10 – 2-3 = 5 โนด
- ระดับที่ 0 -> 20-5-3-2 = 10 โนด

ในการเติมโนดใหม่เข้าไป ให้สุ่มตัวเลขจาก 1 ถึง 20 มา ถ้าตัวเลขนั้นอยู่ระหว่าง 1 ถึง 10 ก็ให้ใส่โนดที่มีลิงค์ระดับที่ 0 ถ้าตัวเลขนั้นอยู่ระหว่าง 11 ถึง 15 ก็ให้ใส่โนดที่มีลิงค์ถึงระดับที่ 1 ถ้าเป็นเลข 16 ถึง 18 ก็ให้ใส่โนดที่มีลิงค์ถึงระดับที่ 2 ขอละไว้ไม่เขียนต่อแล้วนะ

การใส่โนดด้วยวิธีนี้ ในระยะยาวจะสามารถบังคับจำนวนอัตราส่วนของโนดชนิดต่างๆให้คงตัวได้ ส่วนในการเขียนโค้ดภายในของแต่ละโนดนั้น เนื่องจากว่าแต่ละโนดมีจำนวนลิงค์ออกไปต่างกัน วิธีที่ใช้ได้ดีวิธีหนึ่งก็คือ การให้ next เป็นอาร์เรย์ซึ่งเก็บจำนวนลิงค์ได้ต่างๆกันไป

ลิสต์แบบจัดตัวเองได้(Self-Organizing List)

เป็นลิสต์ที่จัดเรียงโนดต่างๆภายในตัวมันเองได้ เพื่อให้สะดวกในการค้นสิ่งของภายใน การโปรแกรมลิสต์แบบนี้มีอยู่หลายวิธี เช่น

- จัดตัวที่เราพึ่งดูข้อมูลเป็นตัวล่าสุดไว้ที่หัวลิสต์ หรือ
- สลับที่โนดที่ดูข้อมูลเป็นตัวล่าสุดกับโนดที่อยู่ข้างหน้ามัน หรือ
- จัดลิสต์โดยให้โนดที่มีสมาชิกถูกดูข้อมูลบ่อยที่สุดไปอยู่หน้าลิสต์ เรียงกันมาตามลำดับหรือ
- กำหนดลักษณะของการเรียงไว้ล่วงหน้า เช่น เรียงข้อมูลตามตัวอักษรในพจนานุกรม วิธีนี้ได้เปรียบตอนหาของในลิสต์ เพราะถ้าหาของ ณ ตำแหน่งหนึ่งไม่เจอ เราจะรู้ว่าของนั้นไม่อยู่ในส่วนที่เหลือของลิสต์แน่ๆ ทำให้ไม่เปลืองเวลาในการหาของต่อ

จากการทดลองที่แสดงในหนังสือของ อดัม ดรอสเด็ค สรุปได้ว่าการจัดตัวที่เราพึ่งดูข้อมูลเป็นตัวล่าสุดไว้ที่หัวลิสต์และการจัดลิสต์โดยให้โนดที่มีสมาชิกถูกดูข้อมูลบ่อยที่สุดไปอยู่หน้าลิสต์นั้นมีความเร็วในระดับเดียวกัน ซึ่งเร็วกว่าวิธีที่สลับโนดที่ดูข้อมูลเป็นตัวล่าสุดกับโนดที่อยู่ข้างหน้ามันและวิธีที่กำหนดลักษณะของการเรียงไว้ล่วงหน้า

ตารางสปาร์ซ(Sparse Table)

ในการใช้ตาราง เช่นอาร์เรย์สองมิติ เก็บข้อมูล ถ้ามีข้อมูลที่เก็บจริงๆอยู่ไม่มากก็จะมีช่องอาร์เรย์ว่างจำนวนมาก ตารางหรืออาร์เรย์สองมิติที่มีที่ว่างจำนวนมากนี้ เราเรียกว่า ตารางสปาร์ซ เราสามารถใช้ลิงค์ลิสต์เก็บข้อมูลแทนตารางแบบนี้ได้ ซึ่งการใช้ลิงค์ลิสต์จะทำให้ประหยัดเนื้อที่ได้มาก ดังตัวอย่างต่อไปนี้

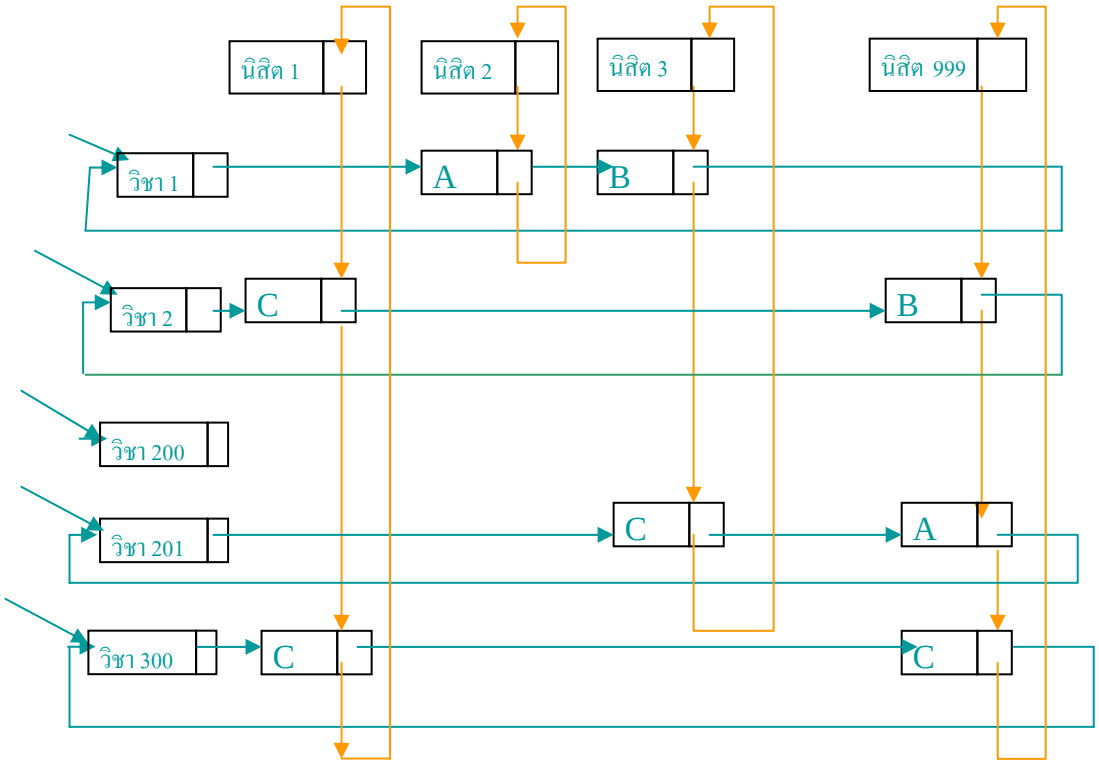
ตัวอย่างที่ 3-1

สมมติว่ามีข้อมูลนิสิตทั้งหมด และวิชาทั้งหมดของมหาวิทยาลัย เราต้องการทำตารางของนิสิตกับวิชา โดยถ้ารู้นิสิต ต้องหาวิชาที่คนนี้ลงทะเบียนได้ และถ้ารู้วิชา ต้องหาได้ว่ามีใครลงวิชานี้บ้าง ถ้าเราใช้อาร์เรย์สองมิติ จะมีช่องว่างอยู่มากอย่างเห็นได้ชัดเพราะนิสิตต่างสาขาไม่ลงของสาขาอื่น ดังนั้นแต่ละวิชาก็จะมีนิสิตลงทะเบียนน้อยมาก ส่วนถ้ามองทางตัวนิสิตเอง นิสิตก็ไม่ไปลงทะเบียนวิชานอกคณะหรือนอกภาควิชามากนัก ทำให้เกิดช่องว่างขึ้นมากได้เช่นเดียวกัน ดังตารางที่ 3.2

ตาราง 3.2 นิสิตกับวิชาที่ลงทะเบียน(บันทึกเกรด)

	นิสิต1	นิสิต2	นิสิต3	…	นิสิต 500	นิสิต 501	นิสิต 502	…	นิสิต 999
วิชา1		A		…			B	…	
วิชา2	C		B	…				…	B
…	…	…	…	…	…	…	…	…	…
วิชา 200				…	A			…	
วิชา 201			C	…				…	A
…	…	…	…	…	…	…	…	…	…
วิชา 300	C			…				…	C

เพื่อการประหยัดที่ เราสามารถใช้ลิงค์ลิสต์เก็บตารางข้างบนได้ดังรูป 3.17 (รูปนี้ไม่ได้บันทึกบางส่วนของตารางไว้ เนื่องจากมีเนื้อที่หน้ากระดาษไม่พอ)

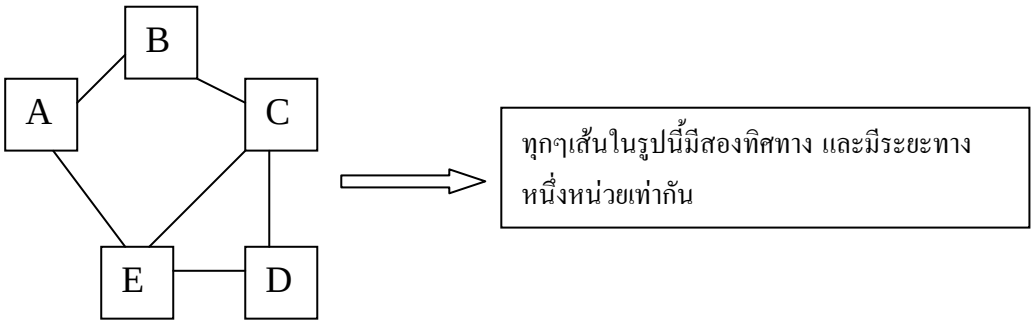


รูป 3.17 Sparse Table ของเกรดนิสิตและวิชาที่ลง

อย่าลืมว่า ถ้าเราใช้อาร์เรย์ ที่ว่างในอาร์เรย์ก็กินหน่วยความจำ ดังนั้นถ้าเราเก็บแค่ส่วนที่ต้องการในลิงค์ลิสต์ก็จะสามารถประหยัดเนื้อที่ในหน่วยความจำได้มาก ตัวอย่างในรูปที่ 3.17 เป็นลิงค์ลิสต์แบบต่อเป็นวงเพื่อให้เราสามารถตรวจสอบชื่อวิชาได้ในกรณีที่เราต้องการหาวิชาที่นิสิตคนหนึ่งลง โดยเราตามลิงค์จากนิสิตคนนั้นไปยังโนดๆหนึ่ง แล้วเปลี่ยนมาตามลิงค์ตามแนวนอนแทนจนกว่าจะเจอชื่อวิชา วิธีตามลิงค์เป็นวงยังทำให้สามารถตรวจสอบชื่อนิสิตที่เรียนวิชาหนึ่งๆได้ด้วย แต่จริงๆการตามลิงค์แบบนี้ค่อนข้างช้า เราอาจทำให้เร็วขึ้นโดยให้แต่ละโนดมีลิงค์กลับไปที่ชื่อนิสิตหรือชื่อวิชาโดยตรง

การเขียนโครงสร้างข้อมูลของกราฟ

วิธีที่ง่ายที่สุดคือการสร้างตารางขึ้นมาแล้วสร้างลิงค์ลิสต์แบบที่ทำกับตารางสปาร์ซ ยกตัวอย่างเช่น ถ้าเรามีกราฟดังรูปที่ 3.18



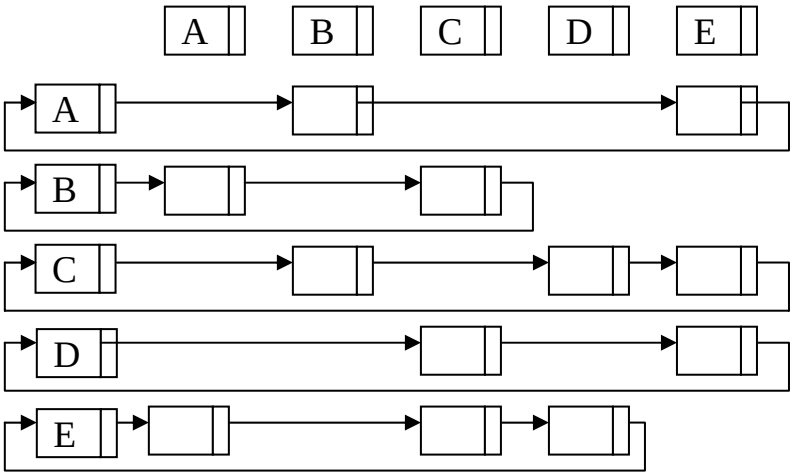
รูป 3.18 ตัวอย่างกราฟ

เราเขียนกราฟนี้เป็นตารางได้ดังนี้

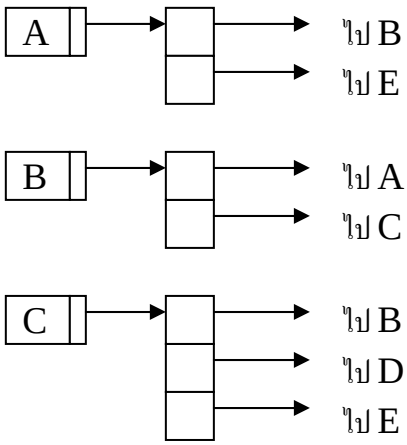
	A	B	C	D	E
A ไปที่	0	1	0	0	1
B ไปที่	1	0	1	0	0
C ไปที่	0	1	0	1	1
D ไปที่	0	0	1	0	1
E ไปที่	1	0	1	1	0

และนำไปสร้างเป็นลิงค์ลิสต์ได้ดังรูปที่ 3.19 จริงๆแล้วในรูปที่ 3.19 นั้น ข้อมูลลิงค์ในแนวตั้งและแนวนอนนั้นเหมือนกัน เราจึงสามารถตัดลิงค์ในแนวหนึ่งทิ้งได้ แต่ถ้าเป็นกราฟลูกศรมีทิศทางเดียว ตารางจะไม่สมมาตร ทำให้ต้องใช้ข้อมูลทั้งสองแกน

นอกจากนี้ยังมีวิธีอื่นที่เป็นที่นิยมใช้ในการเก็บข้อมูลของกราฟ เช่น การใช้โนดไดเร็กทอรี่ หรือพูดง่ายๆคือ ลิงค์ลิสต์ที่ลิงค์ไปยังอาร์เรย์ของโนด ถ้าเราจะแทนกราฟในรูป 3.18 ด้วยการใช้โนดไดเร็กทอรี่ จะได้ผลดังรูปที่ 3.20 (แสดงเฉพาะเส้นทางจาก A B และ C)



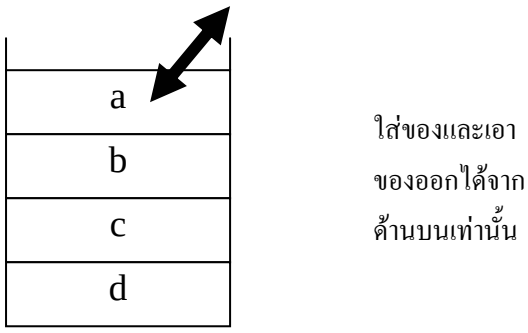
รูป 3.19 สร้างกราฟด้วยแนวคิดของตารางสปาร์ซ เพื่อไม่ให้ดูยุ่งจึงไม่แสดงลิงค์ในแนวตั้ง



รูป 3.20 สร้างกราฟโดยใช้โนดไดเร็กทอรี่

โครงสร้างข้อมูลสแตก(Stack)

เอาล่ะ เรามาต่อกันเลยดีกว่า สแตก (stack) นั้น จริงๆคือถังใส่ของที่ใส่ของทับกันไปเรื่อยๆเป็นชั้นๆ เราจะใส่ของได้ทีละชั้นเท่านั้น เรียกได้ว่าเป็นการใส่ของแบบ last in first out หรือ LIFO การใช้สแตกนั้นง่ายสำหรับระบบคอมพิวเตอร์(ดังจะมีตัวอย่างต่อๆไป) แต่ว่ามีข้อเสียคือ ของที่เอาออกจากสแตกไปแล้วจะหายไปเลย นำมาใช้ต่อไม่ได้ รูป 3.21 แสดงสแตกที่ใส่ d c b และ a ลงไปตามลำดับ



รูป 3.21 สแตก

แล้วเราจะทำอะไรกับสแตกได้บ้าง จริงๆแล้วเราสามารถจัดของเข้าและออกจากสแตกได้ด้วยคำสั่งสามคำสั่งคือ

- Push: เอาของใส่ด้านบนของสแตก
- Pop: เอาสมาชิกบนสุดออกจากสแตก
- Top: บอกเราว่าสมาชิกตัวบนสุดของสแตกคืออะไร แต่ไม่เปลี่ยนสแตก

เราสามารถใช้อาร์เรย์สร้างสแตกขึ้นมาได้ ใช้ลิงค์ลิสต์ทำก็ได้ ก่อนอื่นเรามาดูการใช้ลิงค์ลิสต์ทำสแตกกันก่อนดีกว่า คำสั่งด้านบนจะเทียบได้กับคำสั่งของลิงค์ลิสต์ดังนี้

- Push: เอาของใส่ที่ด้านหน้าลิสต์

- Pop: เอาสมาชิกหน้าสุดออกจากลิสต์ (แค่เลื่อนพอยต์เตอร์ที่บอกส่วนบนของสแตกไปยังโนดถัดไปก็ได้แล้ว)
- Top: รีเทิร์นสมาชิกตัวบนสุดของสแตกแต่ไม่เปลี่ยนสแตก

รูปที่ 3.22 แสดงโค้ดของสแตกที่สร้างด้วยลิงค์ลิสต์

อาร์เรย์ก็สามารถทำสแตกได้ แต่อย่าลืมว่าเราต้องบอกขนาดของอาร์เรย์ล่วงหน้า ซึ่งทำให้เปลืองเนื้อที่ได้ เพื่อความสะดวกในการใช้อาร์เรย์สร้างสแตก เราจะให้มีตัวแปรตัวหนึ่งที่เก็บค่าดัชนีตำแหน่งของอาร์เรย์ที่เก็บสมาชิกตัวบนสุดของสแตกไว้ (ให้ตัวแปรนี้มีค่า -1 ถ้าสแตกว่าง) นี่จะคล้ายกับการที่เราเก็บโนดของลิงค์ลิสต์ที่มีข้อมูลบนสุดของสแตกไว้นั่นเอง ตัวโค้ดของสแตกที่สร้างจากอาร์เรย์นั้นอยู่ในรูปที่ 3.23

การใช้งานสแตก

ใช้ตรวจจำนวนคู่ของวงเล็บในโค้ด

เราสามารถใช้สแตกอ่านตัวโค้ดของโปรแกรม ถ้าเจอวงเล็บเปิดให้เอาใส่สแตก ถ้าเจอวงเล็บปิดให้เอาของออกจากสแตกเสีย ถ้าพยายามเอาของออกจากสแตกแต่ไม่มีอะไรในสแตก แสดงว่าวงเล็บปิดที่เจอนั้นเป็นวงเล็บที่เกินมา เราต้องรายงานระบบ ถ้าวงเล็บที่เอาออกจากสแตกไม่ใช่วงเล็บที่อ่านจากโค้ดก็ถือว่าวงเล็บนั้นอยู่ผิดที่เหมือนกัน และเมื่อตรวจทั้งโปรแกรมเสร็จแล้ว ถ้ายังมีวงเล็บหลงเหลือในสแตกก็ถือว่าเป็นวงเล็บที่เกินมา

ใช้ในการคิดเลขโพสต์ฟิกส์ (postfix expression evaluation)

โพสต์ฟิกส์ คือการเขียนโจทย์เลขโดยนำโอเปอเรเตอร์ไว้หลังตัวเลขที่ถูกกระทำด้วยโอเปอเรเตอร์นั้น เช่น

- 2+3 จะเขียนได้เป็น 2 3 +
- [7+(8*9)+5]*10 จะเขียนได้เป็น 7 8 9 * 5 + + 10 *

เราสามารถใช้สแตกแก้โจทย์โพสต์ฟิกส์โดย

- ถ้าอ่านเจอตัวเลข ให้เอาตัวเลขนั้นใส่สแตก
- ถ้าเจอโอเปอเรเตอร์ ให้เอาตัวเลขที่อยู่ในสแตกออกมาเพื่อใช้งานโอเปอเรเตอร์นั้น แล้วเอาผลการคำนวณใส่กลับเข้าไปในสแตก

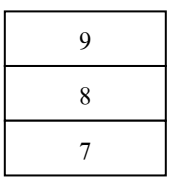
```
1:        public class Stack{
2:          private ListNode top; //ไม่มี header node
3:
4:          public Stack( ){
5:              top = null;
6:          }
7:
8:          /**
9:           * ทดสอบว่าสแตกเต็มหรือไม่ ในที่นี้เราสมมติให้ไม่มีวันเต็ม
10:          * @return false เสมอ
11:          */
12:         public boolean isFull( ){
13:             return false;
14:         }
15:
16:         /**
17:          * ทดสอบว่าสแตกว่างหรือไม่
18:          * @return true ถ้าว่าง มิฉะนั้นให้รีเทิร์น false
19:          */
20:         public boolean isEmpty( ){
21:             return top == null;
22:         }
23:
24:         /**
25:          * ทำให้สแตกว่าง
26:          */
27:         public void makeEmpty( ){
28:             top = null;
29:         }
30:
31:         /**
32:          * รีเทิร์นของที่อยู่บนสุดในสแตก แต่ไม่เปลี่ยนสแตก
33:          * @return ของที่อยู่บนสุดในสแตก (ของชิ้นล่าสุดที่ใส่สแตก)
34:          * @throw Underflow ถ้าสแตกว่าง
35:          */
36:         public Object top( ) throws Underflow{
37:             if( isEmpty( ) )
38:                 throw new Underflow();
39:             return top.element;
40:         }
41:
42:         /**
43:          * เอาสมาชิกบนสุดของสแตกทิ้งไป
44:          * @exception Underflow ถ้าสแตกว่าง
45:          */
46:         public void pop( ) throws Underflow{
```

```
47:             if( isEmpty( ) )
48:                 throw new Underflow( );
49:             top = top.next;
50:         }
51:
52:         /**
53:          * เอาสมาชิกบนสุดออกจากสแตก รีเทิร์นสมาชิกตัวนั้นมาด้วย
54:          * @return ของที่อยู่บนสุดในสแตก
55:          * @exception Underflow ถ้าสแตกว่าง
56:          */
57:         public Object topAndPop( ) throws Underflow{
58:             if( isEmpty( ) )
59:                 throw new Underflow( );
60:
61:             Object topItem = top.element;
62:             top = top.next;
63:             return topItem;
64:         }
65:
66:         /**
67:          * เอาของใหม่ยัดใส่ลงไปในสแตก
68:          * @param x ออบเจ็คต์ที่จะเอาใส่สแตก
69:          */
70:         public void push( Object x ){
71:             top = new ListNode( x, top );
72:         }
73:     }
```

รูป 3.22 โค้ดของสแตกที่สร้างด้วยลิสต์

ต่อไปนี้เป็นสภาพของสแตกเมื่อแก้ปัญหา 7 8 9 * 5 + + 10 *

ตอนแรกอ่าน 7 8 9 เข้ามาในสแตก เพราะฉะนั้นสแตกจะมีสมาชิกภายในดังนี้



ต่อมา อ่านเครื่องหมายคูณ ต้องเอาสิ่งที่อยู่บนสแตกสองตัวออกมาคูณกัน แล้วคืนผลใส่สแตก

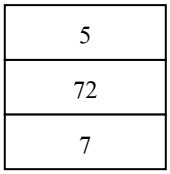
```
1: public class StackFromArray{
2:         private Object[ ] arrayBody;
3:         private int       topIndex;
4:         static final int DEFAULT_CAPACITY = 10;
5:
6:         public StackFromArray( ){
7:             this( DEFAULT_CAPACITY );
8:         }
9:
10:        /**
11:         * สร้างสแตกขึ้นมา โดยให้สแตกที่ว่างมีค่า topIndex เป็น -1
12:         * @param ความจุของสแตกที่สร้างขึ้น
13:         */
14:        public StackFromArray( int capacity ){
15:            arrayBody = new Object[ capacity ];
16:            topIndex = -1;
17:        }
18:
19:        /**
20:         * ทดสอบว่าสแตกว่างไหม จริงๆดูแค่ค่าดัชนี
21:         * @return true ถ้าว่าง false ถ้าไม่ว่าง
22:         */
23:        public boolean isEmpty( ){
24:            return topIndex == -1;
25:        }
26:
27:        /**
28:         * ทดสอบว่าสแตกเต็มไหม ดูจากค่าดัชนี
29:         * @return true ถ้าเต็ม false ถ้าไม่เต็ม
30:         */
31:        public boolean isFull( ){
32:            return topIndex == arrayBody.length - 1;
33:        }
34:
35:        /**
36:         * ทำให้สแตกว่าง จริงๆคือเปลี่ยนค่าดัชนีเท่านั้น
37:         */
38:        public void makeEmpty( ){
39:            topIndex = -1;
40:        }
41:
42:        /**
43:         * @return สมาชิกตัวที่ใส่สแตกไปที่เป็นตัวล่าสุด
44:         * @exception Underflow ถ้าสแตกว่าง
45:         */
46:        public Object top( ) throws Underflow
47:        {
```

```
48:             if( isEmpty( ) )
49:                 throw new Underflow( );
50:             return arrayBody[ topIndex ];
51:         }
52:
53:         /**
54:          * เอาสมาชิกตัวล่าสุดในสแตกทิ้งไป
55:          * @exception Underflow ถ้าสแตกว่าง
56:          */
57:         public void pop( ) throws Underflow{
58:             if( isEmpty( ) )
59:                 throw new Underflow( );
60:             arrayBody[ topIndex-- ] = null;
61:         }
62:
63:         /**
64:          * ใส่สมาชิกใหม่ลงในสแตก
65:          * @param x ของที่เอาใส่
66:          * @exception Overflow ถ้าสแตกเต็ม
67:          */
68:         public void push( Object x ) throws Overflow{
69:             if( isFull( ) )
70:                 throw new Overflow( );
71:             arrayBody[ ++topIndex ] = x;
72:         }
73:
74:         /**
75:          * เอาสมาชิกตัวล่าสุดในสแตกทิ้งไป
76:          * @return สมาชิกตัวที่ทิ้งไปนั้น
77:          * @exception Underflow ถ้าสแตกว่าง
78:          */
79:         public Object topAndPop( ) throws Underflow{
80:             if( isEmpty( ) )
81:                 throw new Underflow( );
82:             Object topItem = top( );
83:             arrayBody[ topIndex-- ] = null;
84:             return topItem;
85:         }
86:}
```

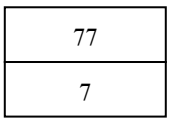
รูป 3.23 โค้ดของสแตกที่สร้างจากอาร์เรย์

ดังนั้นสแตกในตอนนี้จะเป็น

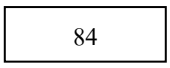
ต่อมาอ่านเลข 5 ก็เอาใส่สแตกเฉยๆ



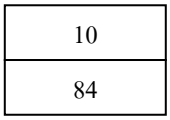
ต่อมาอ่าน + ก็ต้องเอาสมาชิกสองตัวในสแตกมาบวกกันแล้วใส่ผลกลับไป จะได้



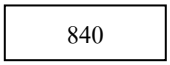
ต่อมาอ่าน + อีกครั้ง คราวนี้ก็เอาสมาชิกสองตัวบนสแตกไปบวกกันเช่นเคย



ต่อมาอ่านเลข 10 ก็เอาใส่ลงไปในสแตกเฉยๆ



สุดท้ายก็คือการคูณ เอาสมาชิกสองตัวในสแตกออกไปคูณกันก็จะได้คำตอบ



การคำนวณพื้นฐานภายในเครื่องคอมพิวเตอร์นั้น จะอยู่ในรูปของการใช้สแตก จาวาเองก็มีสแตกที่ทำงานนี้ เรียกว่าโอเปอรานด์สแตก (operand stack)

ใช้ในการแปลงเลขแบบธรรมดาให้เป็นเลขโพสต์ฟิกส์

เพื่อให้คอมพิวเตอร์สามารถคำนวณเลขได้ จะต้องแปลงเลขที่คนทำกันตามปกติให้อยู่ในรูปของโพสต์ฟิกส์เสียก่อน วิธีการแปลงนั้นมีหลักง่ายๆคือ

- ถ้าอ่านเจอโอเปอรานด์ ให้เอาโอเปอรานด์ตัวนั้นออกไปเป็นส่วนหนึ่งของคำตอบ
- ถ้าเจอโอเปอเรเตอร์ ต้องเอาโอเปอเรเตอร์ที่มีความสำคัญมากกว่าหรือเท่ากันออกจากสแตกไปเป็นส่วนหนึ่งของคำตอบ แล้วให้เอาโอเปอเรเตอร์ตัวใหม่ใส่สแตก
- เก็บเครื่องหมาย “(“ ลงบนสแตกด้วย
- ถ้าเจอเครื่องหมาย “)” ให้เอาของออกจากสแตกไปเป็นส่วนของคำตอบเรื่อยๆ จนกว่าจะเจอ “(“

เอาล่ะ สมมติว่าเรามีเลข a +b*c + (d*e+f) * g เราจะทำให้โจทย์เลขนี้เปลี่ยนไปอยู่ในรูปแบบโพสต์ฟิกส์ได้ดังนี้

ตอนแรกเราอ่าน a เข้ามา เนื่องจากเป็นโอเปอรานด์ เราเอามันออกไปเป็นส่วนหนึ่งของคำตอบเลย จะได้สแตกว่างกับคำตอบที่มี a ตัวเดียว ดังรูปข้างล่าง

สแตก	คำตอบ
ว่าง	a

ต่อมาเราอ่าน + เนื่องจากไม่มีโอเปอเรเตอร์ตัวอื่นในสแตก ฉะนั้นให้เอา + ใส่สแตกไป ต่อจากนั้นก็อ่าน b เข้ามาจากข้อมูลนำเข้า เราเอา b ออกไปเป็นส่วนของคำตอบได้เลย ตอนนี้สแตกและคำตอบจะมีลักษณะดังรูปข้างล่าง

สแตก	คำตอบ
+	a b

ต่อมาอ่าน * เนื่องจากเครื่องหมาย + ที่อยู่บนสแตกมีความสำคัญมากกว่า จึงไม่ต้องเอาออกจากสแตก ดังนั้นเราจึงเอา * ใส่สแตกไปได้เลย ข้อมูลนำเข้าตัวต่อไปคือ c ซึ่งเราสามารถนำไป

เป็นส่วนของคำตอบได้เลย ดังนั้นในตอนที่อ่าน c เข้ามาแล้วนั้น สแตกและคำตอบมีลักษณะดังรูปข้างล่าง

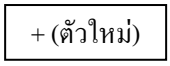
สแตก

คำตอบ

a b c

ต่อมาเป็นการอ่าน + ที่อยู่หลัง c เราต้องเอา + ตัวที่อ่านใหม่นี้ใส่สแตก แต่ว่าต้องเอาโอเปอเรเตอร์ที่มีความสำคัญมากกว่าหรือเท่ากันออกจากสแตกไปเป็นส่วนหนึ่งของคำตอบเสียก่อน ดังนั้น * กับ + ที่อยู่ในสแตกก่อนหน้านี้ต้องถูกเอาออกไป ลักษณะของสแตกและคำตอบในขณะนี้จะเป็น

สแตก

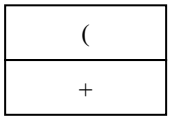


คำตอบ

a b c * +

ตัวถัดไปคือเครื่องหมายวงเล็บเปิด แล้วก็ตามด้วยตัว d เราเอาวงเล็บเปิดใส่สแตกไปเลย ส่วน d ก็เอาเอาท์พุตไปเลย จะได้สแตกและคำตอบดังรูปข้างล่าง

สแตก

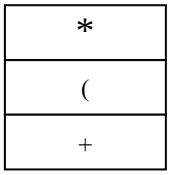


คำตอบ

a b c * + d

ตัวถัดไปที่ต้องอ่านเข้ามาคือ * แล้วก็ตัว e ซึ่งเอาออกไปเอาท์พุตได้เลย ตอนนี้เราจะมีสแตกและคำตอบเป็นดังรูปต่อไปนี้

สแตก

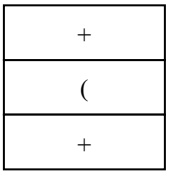


คำตอบ

a b c * + d e

ตัวต่อไปคือ + แล้วก็ f เครื่องหมายคูณที่อยู่ในสแตกมีความสำคัญมากกว่า จึงต้องเอาออกจากสแตกไปเสีย ส่วนเครื่องหมายบวกที่อยู่ด้านล่างของสแตกนั้น แม้จะมีความสำคัญเท่ากับเครื่องหมายบวกตัวใหม่ แต่มันถูกกั้นด้วยวงเล็บ จึงไม่ต้องนำออกไป ส่วนตัว f นั้นนำออกไปเป็นเอาท์พุตได้ตามปกติ

สแตก

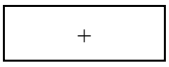


คำตอบ

a b c * + d e * f

ตัวต่อไปคือวงเล็บปิด ตอนนี้เราต้องเอาของในสแตกออกให้หมดจนถึงตรงที่เก็บวงเล็บเปิด

สแตก



คำตอบ

a b c * + d e * f +

ต่อไปที่ต้องอ่านเข้ามาคือ * และ g ซึ่ง * นั้นต้องถูกใส่ในสแตกและ g ก็ต้องถูกนำออกเอาท์พุต

สแตก

คำตอบ

a b c * + d e * f + g

ตอนนี้จะเห็นว่า ตัวอินพุตนั้นถูกอ่านจนหมดแล้ว แต่ยังเหลือของบนสแตก ให้เอาของบนสแตกออกไปต่อกับคำตอบที่สะสมมาให้หมด จะได้คำตอบที่สมบูรณ์ นั่นคือ a b c * + d e * f + g * +

ใช้ในการแปลงเลขธรรมดาให้เป็นพรีฟิกส์

นอกจากโพสต์ฟิกส์แล้ว มีการเขียนเลขอีกประเภทหนึ่งในศาสตร์ทางคอมพิวเตอร์ เรียกว่าการเขียนเลขแบบพรีฟิกส์ ซึ่งเขียนโดยเอาตัวโอเปอเรเตอร์นำหน้าแล้วตามด้วยโอเปอรานด์ที่โอเปอเรเตอร์นั้นทำงาน ตัวอย่างเช่น ถ้าเรามีเลข a+b*c เราจะสามารถเขียนเป็นพรีฟิกส์ได้คือ

+ a * b c

หลักการใช้สแตกในการแปลงเลขธรรมดาให้เป็นพรีฟิกส์นั้นใช้สองสแตก สแตกแรกเก็บโอเปอเรเตอร์ (operator stack) สแตกที่สองเก็บโอเปอรานด์ (operand stack) ส่วนวิธีการทำเป็นดังนี้

ถ้าอ่านเจอตัวเลขจากอินพุต

1. เอาตัวเลขนั้นใส่สแตกของโอเปอรานด์

ถ้าเจอโอเปอเรเตอร์

1. ถ้าสแตกของโอเปอเรเตอร์ว่าง ก็เอาใส่สแตกนั้นซะ
2. ถ้าโอเปอเรเตอร์ที่อ่านเจอคือเครื่องหมายวงเล็บเปิด เอาใส่โอเปอเรเตอร์สแตก
3. ถ้าโอเปอเรเตอร์ที่อ่านเจอมีความสำคัญมากกว่าโอเปอเรเตอร์ที่อยู่บนสุดของโอเปอเรเตอร์สแตก ให้เอาใส่โอเปอเรเตอร์สแตก
4. ถ้าโอเปอเรเตอร์ที่อ่านเจอมีความสำคัญน้อยกว่าหรือเท่ากับโอเปอเรเตอร์ที่อยู่บนสุดของโอเปอเรเตอร์สแตก
    a. ให้เอาโอเปอเรเตอร์ออกจากโอเปอเรเตอร์สแตก
    b. เอาโอเปอรานด์ที่โอเปอเรเตอร์นั้นใช้ออกจากโอเปอรานด์สแตก
    c. เอามาต่อกันแบบโดยให้โอเปอเรเตอร์อยู่ก่อน แล้วตามด้วยโอเปอรานด์โดยเรียงเอาตัวที่ออกมาทีหลังสุดไว้ก่อน
    d. เอาผลลัพธ์ใส่โอเปอรานด์สแตก
5. ถ้าโอเปอเรเตอร์ที่อ่านเจอคือเครื่องหมายวงเล็บปิด หรือเราอ่านจากอินพุตต่อไม่ได้แล้ว ให้ทำตามข้อย่อยของข้อ 4 จนกว่าส่วนบนสุดของโอเปอรานด์สแตกจะเป็นวงเล็บเปิด ต่อจากนั้นก็เอาวงเล็บเปิดนั้นออกทิ้งไปซะ

เรามาดูตัวอย่างกัน สมมติว่าผมต้องการเปลี่ยน –b+sqrt(b*x – d*a*c)/(e*a) ให้เป็นพรีฟิกส์

ตอนแรกเราอ่าน – ,b เข้ามาตามลำดับ จับแต่ละตัวใส่ในสแตกของมัน เราจะได้

operand stack operator stack

b	-

ต่อจากนั้น อ่าน + เข้ามา เนื่องจาก + มีความสำคัญเท่ากับเครื่องหมายที่อยู่บนโอเปอเรเตอร์สแตก ดังนั้นต้องเอาตัวที่อยู่บนสแตกออกมาจัดเรียงกับโอเปอรานด์ของมัน (ในตอนนี้เราต้องรู้ด้วยว่าเครื่องหมายลบที่จะเอาออกจากสแตกต้องการโอเปอรานด์แค่ตัวเดียว ไม่งั้นจะทำไม่ได้) สแตกในตอนนี้จะกลายเป็น

operand stack operator stack

-b	+

ต่อมาอ่านเครื่องหมาย sqrt กับ (เข้ามา sqrt นี้เปรียบเสมือนเป็นเมธอด ดังนั้นจึงมีความสำคัญมากกว่าโอเปอเรเตอร์ทั่วไป เอาใส่สแตกไปเลย ส่วน (ก็เอาใส่สแตกไปได้เลยเช่นกันตามกฎของมัน

operand stack operator stack

-b	(
	sqrt
	+

ลำดับข้อมูลเข้าต่อมาคือ b, *, x เราเอา b ใส่โอเปอรานด์สแตกไปได้เลย ต่อมาก็เอา * ใส่โอเปอเรเตอร์สแตกได้ เพราะมีวงเล็บอยู่ ส่วน x ก็เอาใส่โอเปอรานด์สแตกไปได้เลยเช่นกัน

operand stack

x
b
-b

operator stack

*
(
sqrt
+

ตัวต่อมาที่อ่านจากอินพุตคือ - เนื่องจากมีความสำคัญน้อยกว่า * ดังนั้นต้องมีการป๊อปสแตกเอา * ออกไปทำงาน

operand stack

*bx
-b

operator stack

-
(
sqrt
+

ต่อมาอ่าน d, *, a ซึ่ง * นั้นมีความสำคัญมากกว่า – ดังนั้นจึงเอาใส่สแตกได้เลย ส่วน d กับ a ก็เอาใส่สแตกได้เลยเช่นกัน

operand stack

a
d
*bx
-b

operator stack

*
-
(
sqrt
+

ถัดมาก็อ่าน * เข้ามาอีกตัว คราวนี้เจอ * ตรงบนสุดของสแตกเหมือนกัน ความสำคัญเท่ากัน ดังนั้นต้องป็อปของจากสแตกมาทำงาน

operand stack

*da
*bx
-b

operator stack

* (ตัวใหม่)
-
(
sqrt
+

ต่อมาอ่าน c ซึ่งไม่มีอะไรพิเศษ เอาใส่สแตกเฉยๆ

operand stack

c
*da
*bx
-b

operator stack

* (ตัวใหม่)
-
(
sqrt
+

จากนั้นอ่าน) ตอนนี้ต้องป็อปสแตกเอาโอเปอเรเตอร์กับโอเปอรานด์มาเรียงจนกว่าจะถึงเครื่องหมาย (แล้วก็ลบ (นั้นออก ด้านล่างนี้เราแสดงสภาพของสแตกตอนที่ป็อป * และ – ออกไปตามลำดับ ก่อนอื่นเอา * ออกไปทำงานกับโอเปอรานด์สองตัวของโอเปอรานด์สแตก

operand stack

**dac
*bx
-b

operator stack

-
(
sqrt
+

ต่อไปก็ปีอป – ออกไปทำงานบ้าง แล้วเอาวงเล็บเปิดทิ้งไป

operand stack

-*bx**dac
-b

operator stack

sqrt
+

ต่อมาอ่านเครื่องหมาย / ซึ่งสำคัญน้อยกว่า sqrt (เราถือว่าเมธอดสำคัญกว่าเครื่องหมายทั่วไป) ดังนั้นต้องเอา sqrt ออกมาทำงาน

operand stack

sqrt - *bx**dac
-b

operator stack

/
+

ต่อมาอ่าน (และ e ซึ่งเอาใส่สแตกไปได้เลย

operand stack

e
sqrt - *bx**dac
-b

operator stack

(
/
+

ต่อมาอ่าน * กับ a ซึ่งเอาใส่สแตกได้ทั้งคู่ ตัวเครื่องหมายบนสแตกนั้นคือวงเล็บ ซึ่งเอามาเทียบกับ * ไม่ได้

operand stack

a
e
sqrt - *bx**dac
-b

operator stack

*
(
/
+

ต่อมาอ่าน) ตอนนี้ก็ป๊อป * ออกไปทำงานจากนั้นก็ทิ้งตัววงเล็บไป

operand stack

*ea
sqrt - *bx**dac
-b

operator stack

/
+

ตอนนี้การอ่านจากอินพุตได้สิ้นสุดลงแล้ว ที่เหลือก็ป๊อปโอเปอเรเตอร์ไปทำงานกับโอเปอรานด์เรื่อยๆ ก่อนอื่นป๊อป / ซะก่อน

operand stack

/ sqrt - *bx**dac*ea
-b

operator stack

+

สุดท้ายก็ป๊อป + ไปทำงาน จะได้ผลลัพธ์อยู่บนโอเปอรานด์สแตก

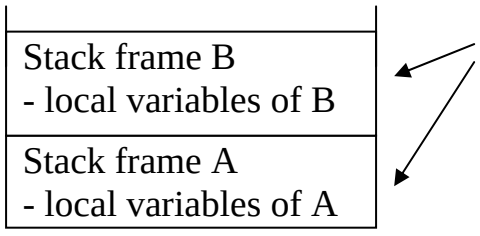
operand stack

+ -b/ sqrt - *bx**dac*ea

operator stack

ใช้ในการเรียกใช้เมธอดของภาษาอย่างจาวา

ข้อมูลที่เมธอดนำมาใช้งานนั้น จาวาจัดเก็บอยู่ในโครงสร้างข้อมูลสแตก โดยข้อมูลของเมธอดหนึ่งจะถือเป็นสมาชิกตัวหนึ่งของสแตก เราเรียกข้อมูลของเมธอดหนึ่งที่เก็บในสแตกนี้ว่า สแตกเฟรม (Stack Frame) หรือ แอกติเวชั่นเรคอร์ด (Activation Record) ยามที่เราเรียกใช้เมธอดในภาษาจาวา หน่วยความจำภายในจาวาจะสร้างสแตกเฟรมใหม่ขึ้นและข้อมูลตัวแปรกับสิ่งต่างๆที่ใช้งานตอนที่เมธอดใหม่นี้ทำงาน (เช่น เมธอดพารามิเตอร์) ก็จะถูกเก็บไว้ในสแตกเฟรมนี้ รูปที่ 3.24 แสดงสแตกเมื่อเมธอด A เรียกใช้เมธอด B สแตกเฟรมจะถูกใช้งานจนกว่าเมธอดใหม่นี้จะสิ้นสุดการทำงาน ซึ่งเมื่อเมธอดนี้สิ้นสุดการทำงาน สแตกเฟรมของเมธอดนี้ก็จะถูกลบทิ้งไปเพื่อให้หน่วยความจำส่วนนี้ถูกนำมาใช้ได้ใหม่



โปรแกรมมองเห็นทั้งตัวแปรจาก A และ B แต่เมื่อ B สิ้นสุดการทำงาน เฟรมของ B จะถูกทำลาย ทำให้เห็นแต่ตัวแปรของ A อย่างเดียว

รูป 3.24 สแตกเฟรมจากการเรียกใช้เมธอด

จะเห็นว่า เมื่อมีการเรียกใช้เมธอดครั้งหนึ่งก็ต้องมีการจองที่ในหน่วยความจำสำหรับเก็บข้อมูลของเมธอดนั้น ดังนั้นการเรียกใช้เมธอดมากๆจะเป็นการสิ้นเปลืองหน่วยความจำ โดยเฉพาะเมื่อมีการเรียกใช้เมธอดที่ไม่มีความจำเป็นจะต้องเก็บข้อมูลลงสแตก เราเรียกเมธอดแบบนี้ว่า เทลรีเคอร์ชั่น (Tail Recursion) ตัวอย่างของเทลรีเคอร์ชั่นอยู่ในโค้ดของรูปที่ 3.25

```
1:    public static void myProgram(ListNode p, Object x){
2:        if (p == null)
3:             return;
4:        p.element =  x;
5:        myProgram(p.next, x);
6:    }
```

รูป 3.25 โปรแกรมที่เป็น tail recursion

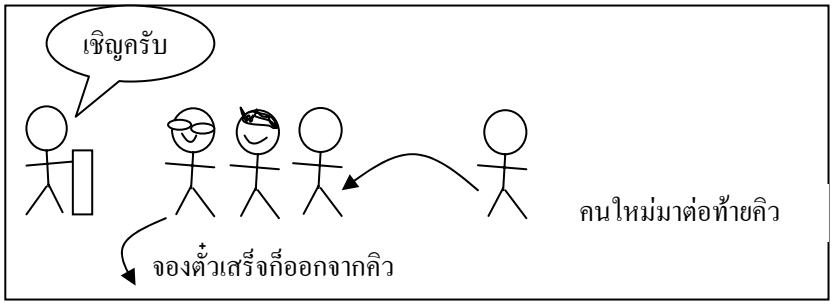
จะเห็นว่า โปรแกรมนี้เก็บตัวแปร p กับ x เอาไว้ใช้งาน แต่เมื่อมีการเรียกเมธอดใหม่ในบรรทัดสุดท้าย ปรากฏว่าหลังจากนี้เมธอดเดิมจะไม่มีการคำนวณอะไรต่อแล้ว ดังนั้นของที่อยู่ในสแตกก็ไม่มีความจำเป็นที่ต้องเก็บต่อไป แต่เพราะเมธอดเดิมนั้นยังไม่รีเทิร์น จึงเก็บสแตกเอาไว้อย่างนั้น ทำให้เปลืองหน่วยความจำ คอมไพเลอร์ที่ดีจะดูออกและสามารถลบสแตกเฟรมที่ไร้ประโยชน์ได้ แต่ตามปกติเราจะไม่ทราบว่าคอมไพเลอร์จะรู้หรือไม่ ดังนั้นทางที่ดีที่สุดที่จะประหยัดหน่วยความจำก็คือ ยามที่เราเจอเทลรีเคอร์ชั่น ให้เปลี่ยนให้เป็นการใช้ลูปเสีย รูป 3.26 แสดงโปรแกรมจากรูป 3.25 ที่แปลงเป็นลูปแล้ว

```
1:    public static void myProgram(ListNode p, Object x){
2:          ListNode temp = p;
3:          while (true){
4:             if (temp == null)
5:                  return;
6:             temp.element =  x;
7:             temp = temp.next;
8:          }
9:    }
```

รูป 3.26 การใช้ลูปแทน tail recursion

โครงสร้างข้อมูลแถวคอย หรือคิว(Queue)

คิว จริงๆแล้วก็คือลิสต์นั่นเอง เพียงแต่การเอาของใส่ทำได้แค่ที่ด้านท้าย และการเอาของออกทำได้แค่ที่ด้านหน้า เราเรียกการใส่ของอย่างนี้ว่า first-in first-out (FIFO) เรามีตัวอย่างคิวในชีวิตประจำวันมากมาย เช่นการต่อคิวซื้อตั๋วหนัง ดังรูปที่ 3.27



รูป 3.27 คนต่อคิว

การสร้างคิวขึ้นมามีหลายวิธี เราอาจใช้ลิสต์ก็ได้ หรือ ใช้อาร์เรย์ก็ได้ ถ้าใช้ลิสต์เราก็มีหลักการทำง่ายๆว่าคิวคือลิสต์ที่จำกัดการใส่สมาชิกที่ท้ายลิสต์และการเอาสมาชิกออกที่หัวลิสต์ ดังนั้นคิวมีเมธอดดังนี้

- void enqueue(Object o): เอา O ใส่ลงไปที่ท้ายคิว
- Object dequeue(): เอาของที่หัวคิวทิ้งไป รีเทิร์นของนั้นเป็นคำตอบ แจ้งความผิดพลาดถ้าพยายามเอาของจากคิวที่ว่างอยู่

โค้ดของคิวที่สร้างจากลิงค์ลิสต์อยู่ในรูปที่ 3.28 จะเห็นว่าเราไม่ได้ extends ลิงค์ลิสต์ การ extends นั้นมีข้อเสียคือเราได้เมธอดทั้งหมดของลิงค์ลิสต์มา ทำให้เราสามารถเติมสมาชิกตรงไหนของคิวก็ได้ หรือเอาสมาชิกออกจากตำแหน่งไหนก็ได้แบบเดียวกับที่ทำกับลิสต์ พูดง่ายๆคือ การ extends ทำให้เสียความเป็นคิว (นี่ก็เป็นเหตุผลที่ทำให้ผมไม่ extends จากลิงค์ลิสต์ตอนที่ทำสแตก)

ขอให้สังเกตเมธอด findlast() ในรูปที่ 3.28 นี้เทียบกับเมธอด findPrevious(Object x) และ find(Object x) ในรูปที่ 3.9 จะเห็นว่าในรูปที่ 3.9 เราสำรวจลิสต์โดยไล่สำรวจจากโนดโดยตรง

แต่ใน findlast() ของรูปที่ 3.28 นี้เราสำรวจลิสต์โดยใช้อิเทอเรเตอร์ ซึ่งจะเห็นว่าใช้ได้ทั้งสองวิธี แต่ว่าโครงสร้างของอิเทอเรเตอร์นั้นสร้างมาเพื่อใช้สำรวจลิสต์อยู่แล้ว ดังนั้นการที่รูป 3.9 ไม่ได้ใช้อิเทอเรเตอร์ในการสำรวจลิสต์ก็เป็นการไม่ใช้อิเทอเรเตอร์ให้เกิดประโยชน์อย่างเพียงพอ

ในการสร้างคิวจากอาร์เรย์ก็ยังมีเมธอด enqueue กับ dequeue เหมือนเดิม แล้วก็ควรมีส่วนอื่นเพิ่มเติมดังนี้

- front: ดัชนีบอกว่าหัวคิวอยู่ตรงไหน
- back: ดัชนีบอกว่าสมาชิกตัวท้ายคิวอยู่ตรงไหน
- size: ตัวเลขบอกจำนวนสมาชิกในอาร์เรย์

การ enqueue(Object o) จะทำได้ด้วยหลักดังนี้

- size++
- back++ (เป็นการเตรียมที่ด้านท้ายคิวสำหรับใส่สมาชิกใหม่)
- array[back] = o

ส่วนการ dequeue() ก็ทำได้โดย

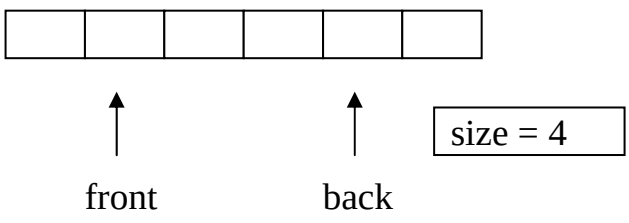
- size - -
- เก็บสมาชิกตำแหน่งนี้ไว้รีเทิร์น
- front++ (เลื่อนสมาชิกตัวหน้าออกไปจากคิว)

รูป 3.29 แสดงภาพของคิวที่ใช้อาร์เรย์สร้าง ที่เราไม่ให้คิวเริ่มต้นจากสมาชิกที่มีค่าดัชนีเป็นศูนย์เสมอก็เพราะว่าถ้าใช้วิธีนั้นในการเอาของออกจากคิวแต่ละครั้งจะต้องเสียเวลาเลื่อนสมาชิกในอาร์เรย์ทุกตัว

```
1:    public class QueueList{
2:    private ListNode header;
3:
4:      public QueueList( ){
5:        header = new ListNode( null );
6:      }
7:
8:      /**
9:       * หาว่าคิวว่างหรือไม่
10:      * @return true ถ้าไม่มีสมาชิกในคิว และ false ถ้ามีสมาชิก
11:      */
12:     public boolean isEmpty(){
13:        return header.next == null;
14:     }
15:
16:     /**
17:      * ทำให้คิวว่างในทันที
18:      */
19:     public void makeEmpty( ){
20:         header.next = null;
21:     }
22:
23:     /**
24:      * หาตำแหน่งของโนดสุดท้าย
25:      * @return อิเทอเรเตอร์บอกตำแหน่งโนดสุดท้าย
26:      * @exception Underflow ถ้าคิวว่างอยู่
27:      */
28:     public LinkedListItr findlast() throws Underflow{
29:       if(isEmpty())
30:         throw new Underflow();
31:       LinkedListItr itr = new LinkedListItr(header);
32:       LinkedListItr itr2 =
33:                         new LinkedListItr(header.next);
34:       while( !itr2.isPastEnd( )){
35:         itr.advance();
36:         itr2.advance();
37:       }
38:       return itr;
39:     }
40:
41:     /**
42:      * บอกว่าสมาชิกตัวที่อยู่ในหัวคิวคืออะไร เมธอดนี้ไม่เปลี่ยนอะไรบนคิวเลย
43:      * @return สมาชิกตัวที่อยู่ในหัวคิว
44:      * @exception Underflow ถ้าคิวว่าง
45:      */
46:     public Object getFront( ) throws Underflow{
```

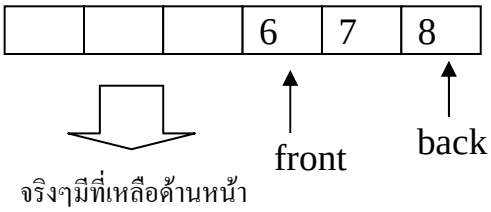
```
47:            if( isEmpty( ) )
48:                throw new Underflow();
49:            return header.next.element;
50:         }
51:
52:       /**
53:        * เอาของยัดใส่ท้ายคิว
54:        * @param o ของที่จะใส่คิว
55:        */
56:       public void enqueue(Object o){
57:         if(isEmpty()){
58:           header.next = new ListNode(o);
59:         }else{
60:           LinkedListItr itr = findLast();
61:           itr.current.next = new ListNode(o);
62:         }
63:       }
64:
65:       /**
66:        * เอาของออกจากหัวคิว
67:        * @return ของที่เอาออกจากหัวคิวนั้น
68:        * @exception Underflow ถ้าคิวว่างอยู่
69:        */
70:       public Object dequeue() throw Underflow{
71:         if(isEmpty()){
72:           throw new Underflow();
73:         }else{
74:            Object item = header.next.element;
75:            header.next = header.next.next;
76:            return item;
77:         }
78:       }
79:    }
```

รูป 3.28 โค้ดของคิวที่สร้างจากลิงค์ลิสต์

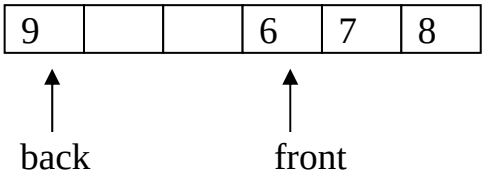


รูป 3.29 คิวที่สร้างจากอาร์เรย์

แต่การสร้างคิวโดยใช้อาร์เรย์นี้ก็มีข้อควรคิดหลายประการ อย่างแรกคืออาจเกิดสถานการณ์ที่มีที่ในอาร์เรย์แต่ว่าเติมสมาชิกไม่ได้เพราะเข้าใจว่าที่เต็ม ดังรูปที่ 3.30 เติมสมาชิกอีกตัวหนึ่งไม่ได้ เพราะเลขแปดอยู่ที่ท้ายอาร์เรย์แล้ว วิธีแก้ปัญหานี้คือดัดแปลงดัชนีของอาร์เรย์ให้สามารถวนกลับไปที่หัวอาร์เรย์ได้ ดังรูปที่ 3.31



รูป 3.30 กรณีที่เติมสมาชิกในคิวไม่ได้เพราะอาร์เรย์ถึงตำแหน่งสุดท้าย



รูป 3.31 ดัชนีของอาร์เรย์เมื่อดัดแปลงให้วนตามขนาดอาร์เรย์

ยังมีข้อสังเกตคือ เมื่อ back มีค่าเท่ากับ front –1 ตัวคิวจะมีสถานะที่เป็นไปได้สองแบบ คือ คิวว่าง กับ คิวเต็ม ดังนั้นเมื่อมีสถานการณ์แบบนี้เกิดขึ้นจึงต้องดู size ประกอบไปด้วย โค้ดของคิวที่สร้างจากอาร์เรย์อยู่ในรูป 3.32

คิวแบบสองหน้า (double-ended queue)

เป็นคิวที่เอาสมาชิกเข้าและออกได้ทั้งที่หัวคิวและท้ายคิว เมธอดที่ควรมีสำหรับคิวแบบนี้นั้นเป็นดังนี้

- insertFirst(Object o): เอา o ใส่ข้างหน้าคิว

```
1:    public class QueueArray{
2:        private Object [ ] theArray;
3:        private int        size;
4:        private int        front;
5:        private int        back;
6:        static final int DEFAULT_CAPACITY = 10;
7:
8:        public QueueArray( ){
9:            this( DEFAULT_CAPACITY );
10:       }
11:
12:       public QueueArray( int capacity ){
13:           theArray = new Object[ capacity ];
14:           makeEmpty( );
15:       }
16:
17:       /**
18:        * ทดสอบว่าคิวว่างหรือไม่
19:        * @return true ถ้าคิวว่าง ถ้าไม่งั้นก็รีเทิร์น false
20:        */
21:       public boolean isEmpty( ){
22:           return size == 0;
23:       }
24:
25:       /**
26:        * ทดสอบว่าคิวเต็มแล้วหรือยัง
27:        * @return true ถ้าเต็ม  ไม่งั้นก็รีเทิร์น false
28:        */
29:       public boolean isFull( ){
30:           return size == theArray.length;
31:       }
32:
33:       /**
34:        * ทำให้คิวว่างไนทันที
35:        */
36:       public void makeEmpty( ){
37:           size = 0;
38:           front = 0;
39:           back = -1;
40:       }
41:
42:       /**
43:        * บอกว่าสมาชิกตัวที่อยู่ในหัวคิวคืออะไร เมธอดนี้ไม่เปลี่ยนอะไรบนคิวเลย
44:        * @return สมาชิกตัวที่อยู่ในหัวคิว
45:        * @exception Underflow ถ้าคิวว่าง
46:        */
47:       public Object getFront( ) throws Underflow{
```

```
48:             if( isEmpty( ) )
49:                 throw new Underflow();
50:             return theArray[ front ];
51:         }
52:
53:         /**
54:          * เอาของใส่ท้ายคิว
55:          * @param x ของที่จะใส่คิว
56:          * @exception Overflow ถ้าคิวเต็ม
57:          */
58:         public void enqueue( Object x ) throws Overflow{
59:             if( isFull( ) )
60:                 throw new Overflow( );
61:             back = increment( back );
62:             theArray[ back ] = x;
63:             size++;
64:         }
65:
66:         /**
67:          * เอาของออกจากหัวคิว
68:          * @return ของที่เอาออกจากหัวคิวนั้น
69:          * @exception Underflow ถ้าคิวว่าง
70:          */
71:         public Object dequeue( )throws Underflow{
72:             if( isEmpty( ) )
73:                 throw new Underflow();
74:             size--;
75:             Object frontItem = theArray[ front ];
76:             theArray[ front ] = null;
77:             front = increment( front );
78:             return frontItem;
79:         }
80:
81:         /**
82:          * เพิ่มค่าดัชนีของอาร์เรย์ วนเลขมาหัวอาร์เรย์ด้วยถ้าสุดอาร์เรย์แล้ว
83:          * @param x ดัชนีของอาร์เรย์
84:          * @return x+1 แต่ถ้า x+1เลยตัวอาร์เรย์ไป ให้รีเทิร์น 0
85:          */
86:         private int increment( int x ){
87:             if( ++x == theArray.length )
88:                 x = 0;
89:             return x;
90:         }
91:     }
```

รูป 3.32 คิวที่สร้างจากอาร์เรย์

- insertLast(Object o): เอา o ใส่ข้างหลังคิว
- removeFirst(): เอาของออกจากหัวคิว รีเทิร์นของนั้นด้วย (แจ้งข้อผิดพลาดถ้าคิวว่าง)
- removeLast(): เอาของออกจากท้ายคิว รีเทิร์นของนั้นด้วย (แจ้งข้อผิดพลาดถ้าคิวว่าง)
- first(): รีเทิร์นของที่อยู่หัวคิว (แจ้งข้อผิดพลาดถ้าคิวว่าง)
- last(): รีเทิร์นของที่อยู่ท้ายคิว (แจ้งข้อผิดพลาดถ้าคิวว่าง)
- size(): รีเทิร์นจำนวนสมาชิกในคิว
- isEmpty(): ทดสอบว่าคิวว่างหรือไม่

เราอาจสร้างคิวแบบสองหน้าขึ้นได้จากลิงค์ลิสต์แบบสองทาง เราอาจให้มีโนดปลอมที่ไม่เก็บสมาชิกอะไรทั้งที่หัวและท้ายลิสต์ การมีโนดปลอมชี้ที่ท้ายลิสต์ด้วยจะทำให้เราประหยัดเวลาในการหาท้ายลิสต์นั่นเอง การสร้างเมธอดต่างๆก็ใช้หลักการย้ายพอยต์เตอร์แบบเดียวกับลิงค์ลิสต์แบบสองทาง

เมธอดของคิวแบบสองหน้านั้นครอบคลุมเมธอดของทั้งสแตกและคิวเลยทีเดียว ตาราง 3.3 และ 3.4 แสดงเมธอดของคิวแบบสองหน้าที่สามารถนำไปเป็นเมธอดของสแตกและคิวแบบธรรมดา

ตาราง 3.3 เมธอดของสแตกที่สร้างจากเมธอดของคิวแบบสองหน้าได้

สแตก	คิวสองหน้า
size()	size()
isEmpty()	isEmpty()
top()	last()
push(x)	insertLast(x)
pop()	removeLast()

ตาราง 3.4 เมธอดของคิวแบบธรรมดาที่สร้างจากเมธอดของคิวแบบสองหน้าได้

คิว	คิวสองหน้า
size()	size()
isEmpty()	isEmpty()
getFront()	first()
enqueue(x)	insertLast(x)
dequeue()	removeFirst()

การประยุกต์ใช้คิว

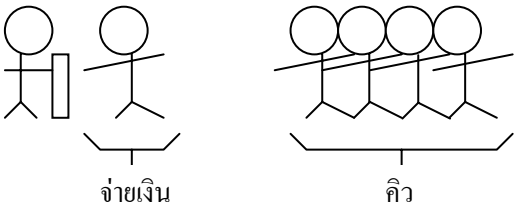
ตัวอย่างที่เห็นชัดน่าจะเป็นการเก็บข้อมูลของออบเจ็กต์ในจาวาเอง สมมุติว่าคลาสของเรามีตัวแปรชื่อ myVar (นิยามอยู่นอกเมธอดใดๆ) มีเมธอดของเราสร้างออบเจ็กต์ขึ้นมาคือ

```
MyObject myVar = new MyObject();
```

แม้ว่าเมธอดที่สร้างออบเจ็กต์นี้ขึ้นมาจะจบการทำงานไปแล้ว แต่ออบเจ็กต์นี้ก็ยังคงอยู่ เพราะว่าตัวแปรถูกนิยามไว้นอกเมธอด ฉะนั้นเราไม่สามารถเก็บข้อมูลของออบเจ็กต์ในเมธอดสแตกได้ จาวาใช้เมโมรี่ฮีป (memory heap) ในการเก็บข้อมูลแบบนี้ เราสามารถใช้คิวในการจัดการเมโมรี่ฮีป ถ้าเราสร้างคิวของเมโมรี่บล็อกที่ไม่ได้ใช้งานเอาไว้ เมื่อมีการสร้างออบเจ็กต์ใหม่เกิดขึ้น สิ่งที่ต้องทำก็คือ dequeue บล็อกหนึ่งออกมาใช้งาน และพอบล็อกนั้นใช้งานเสร็จสิ้นแล้วก็ enqueue กลับเข้าไป

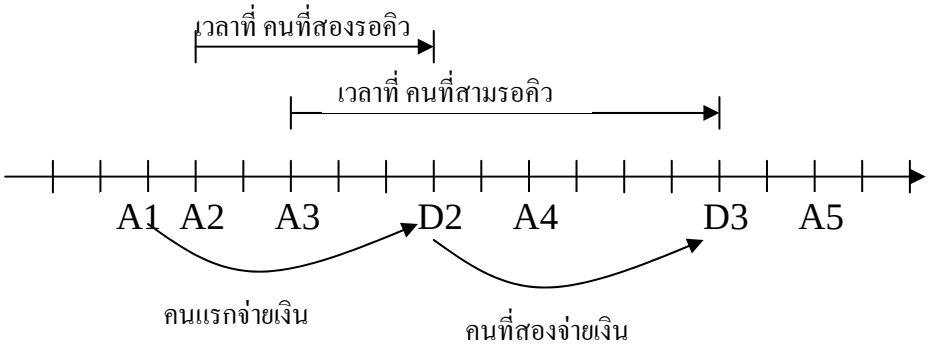
ตัวอย่างที่สองคือการจำลองคิวการให้บริการของร้านค้า ก่อนที่จะสร้างซุปเปอร์มาร์เก็ตต่างๆ นั้นจะต้องมีการคำนวณว่าจำนวนของเคาน์เตอร์ที่ชำระเงินจะเพียงพอกับจำนวนลูกค้าหรือไม่ ลูกค้าจะต้องใช้เวลาในการรอโดยเฉลี่ยเท่าใด การสร้างสถานการณ์จำลองด้วยโปรแกรมคอมพิวเตอร์ก็จะช่วยในการทำนายสิ่งเหล่านี้ได้ คิวก็สามารถถูกนำไปเป็นส่วนหนึ่งของสถานการณ์จำลองนี้ได้ โดยจำลองคิวที่คนต่อซื้อของจริงๆ

เรามาลองดูตัวอย่างง่ายๆกันถึงการจำลองร้านขายของชำซึ่งมีเคาน์เตอร์อยู่หนึ่งเคาน์เตอร์เท่านั้น ให้ลูกค้าที่ยังไม่ได้รับบริการรออยู่ในคิวดังรูปที่ 3.33 ในที่นี้เราต้องการหาว่าลูกค้าใช้เวลารออยู่ในคิวโดยเฉลี่ยเท่าใด สมมติว่าเราใช้เวลาคิดเงินลูกค้าแต่ละคนเป็นเวลา 60 วินาที ถ้ามีคนเข้ากับออกจากคิวพร้อมกัน ให้ทำการออกจากคิวก่อน ถ้าไม่มีคนอยู่ในคิว ให้ลูกค้าที่มาใหม่จ่ายเงินได้เลย เราจะถือว่าลูกค้าคนที่คนขายคิดสตางค์อยู่นั้นได้ออกจากคิวแล้ว



รูป 3.33 คิวของเคาน์เตอร์ขายของ

ต่อไปก็ต้องมีข้อมูลอัตราการมาซื้อของของลูกค้า ซึ่งข้อมูลนี้สามารถสร้างขึ้นได้ด้วยหลักสถิติ ซึ่งอยู่นอกเหนือขอบเขตของหนังสือเล่มนี้ ดังนั้นผมจะขอใช้ข้อมูลที่สร้างขึ้นมาเอง ให้ลูกค้ามาที่เคาน์เตอร์ ณ เวลาดังต่อไปนี้นับจากโปรแกรมเริ่มทำงาน นั่นคือ 30, 40, 60,110, 170 ซึ่งถ้าเราวาดเส้นเวลา โดยแต่ละช่องแทนสิบวินาที จะได้ดังรูป 3.34



รูป 3.34 เส้นเวลาของเหตุการณ์ในการรอคิวซื้อของ

ตัว A ในรูป แสดงถึงการมาที่เคาน์เตอร์ ส่วนตัว D แสดงถึงการออกจากคิว A1 นั้นไม่ต้องรอคิวเพราะเป็นคนแรก ส่วน A2 นั้นต้องรอคิวเพราะต้องรอให้ A1 จ่ายเสร็จเสียก่อน A3 นั้นยิ่งต้อง

รอใหญ่ เพราะ ต้องรอทั้งคนที่หนึ่งและคนที่สองจ่ายเงิน โค้ดจะอยู่ในรูปแบบอย่างคร่าวๆ ดังรูปที่ 3.35

จริงๆแล้ว เราจะรู้แต่เวลาที่คนมาที่เคาน์เตอร์เท่านั้น ดังนั้นโค้ดของเมธอดเมนจึงทำการรับข้อมูล nextArrivalTime เท่านั้น โดยลูปรับข้อมูลการมาของคนเรื่อยๆ แล้วเมื่อไม่มีการมาต่อคิวแล้วก็จัดการการออกจากคิวให้เรียบร้อย

เมธอดที่สำคัญจริงๆคือ processArrival() และ processDeparture() เมธอด processArrival() นั้นถูกเรียกใช้เมื่อมีคนพร้อมจะจ่ายเงินที่เคาน์เตอร์ โดยถ้าไม่มีคนจ่ายเงินอยู่ที่เคาน์เตอร์ ก็ให้ไปจ่ายเงินได้เลย ซึ่งเหตุการณ์นี้จะเป็นการตั้งเวลาที่ลูกค้าคนใหม่จะจ่ายเงินเสร็จโดยปริยาย ส่วนในกรณีที่มีคนกำลังจ่ายเงินอยู่ที่เคาน์เตอร์ก็ให้เอาคนใหม่ใส่คิวรอไป

ส่วนเมธอด processDeparture() นั้น จะถูกเรียกเมื่อลูกค้าคนหนึ่งจ่ายเงินที่เคาน์เตอร์เสร็จแล้ว โดยถ้าคิวที่รอยังมีคนก็ให้เอาคนที่รอออกจากคิว คำนวณเวลาที่คนที่ออกจากคิวรอคิว และคำนวณเวลาในการรอคิวของส่วนรวม รวมทั้งต้องคำนวณเวลาที่คนใหม่นี้จะจ่ายเงินเสร็จด้วย

```
1:    class Simulation{
2:        static int INFINITY = 1000;
3:        static int SERVE_TIME = 60; //เวลาคิดเงินลูกค้าหนึ่งคน
4:        int nextArrivalTime; //เวลาต่อไปที่จะมีลูกค้ามาเตรียมจ่ายเงิน
5:        int currentTime;
6:        int nextDepartureTime; //เวลาต่อไปที่จะมีลูกค้าที่จ่ายเงินเสร็จสิ้น
7:        int numberOfCustomers;
8:        int waitingTime;
9:        int sumOfWaitingTime;
10:       QueueList customerQueue;
11:
12:       public Simulation(){
13:           nextArrivalTime = 0;
14:           currentTime = 0;
15:           nextDepartureTime = INFINITY; //แสดงว่าไม่มีลูกค้า
16:           numberOfCustomers = 0;
17:           waitingTime = 0;
18:           sumOfWaitingTime = 0;
19:           customerQueue = new QueueList();
20:
21:       }
22:
23:       public void processArrival(){
24:         currentTime = nextArrivalTime;
25:         numberOfCustomers++;
26:         if(nextDepartureTime == INFINITY){
27:          nextDepartureTime = currentTime + SERVE_TIME;
28:         } else {
29:          customerQueue.enqueue(new
30:                        Customer(nextArrivalTime));
31:       }
32:
33:       public void processDeparture(){
34:         currentTime = nextDepartureTime;
35:         if(!customerQueue.isEmpty()){
36:           Customer c =
37:             (Customer)customerQueue.dequeue();
38:           waitingTime =
39:              currentTime - c.getArrivalTime();
40:           sumOfWaitingTime += waitingTime;
41:           nextDepartureTime = currentTime + SERVE_TIME;
42:         } else{
43:           // กรณีนี้คือไม่มีลูกค้าแล้ว
44:           waitingTime = 0;
45:           nextDepartureTime = INFINITY;
46:       }
47:
```

```
48:       public void process(int time){
49:           nextArrivalTime = time;
50:           while(nextDepartureTime <= nextArrivalTime)
51:           //ถ้ายังมีการออกจากคิวที่เกิดก่อนเวลาลูกค้าใหม่มา
52:               processDeparture();
53:           processArrival();
54:       }
55:
56:       public void calculateMeanWaitingTime(){
57:           double r = (double)sumOfWaitingTime /
58:                           numberOfCustomers;
59:           System.out.println(r);
60:       }
61:
62:       public static void main(String[] args){
63:            Simulation a = new Simulation();
64:           while(true){
65:            if(ไม่มีข้อมูลของการมาที่เคาน์เตอร์แล้ว)
66:                break;
67:            a.process(ข้อมูลเวลาการมาที่เคาน์เตอร์อันที่อ่านมาได้ใหม่);
68:           }
69:
70:           while(a.nextDepartureTime < INFINITY){
71:           //เมื่อยังมีลูกค้าอยู่ในคิว
72:              a.processDeparture();
73:           }
74:
75:           //ถึงตอนนี้ทุกอย่างได้ทำมาทั้งหมดแล้ว เหลือแต่คำนวณค่าเวลาในการรอโดยเฉลี่ย
76:           a.calculateMeanWaitingTime();
77:       }
78:   }
```

รูป 3.35 การจำลองคิวในร้านขายของเพื่อวิเคราะห์เวลาในการรอคิว

แบบฝึกหัด

1. จงเขียนเมธอด public LinkedList append(LinkedList a, LinkedList b) ทำการต่อลิงค์ลิสต์สองตัวเข้าด้วยกัน

2. จงสร้างเมธอด LinkedList union(LinkedList a, LinkedList b) และ LinkedList intersect(LinkedList a, LinkedList b)ขึ้นมาเพื่อยูเนี่ยนและอินเตอร์เซ็กลิสต์สองลิสต์เข้าด้วยกัน

3. จงเขียนเมธอด LinkedList difference(LinkedList a, LinkedList b) เพื่อสร้างลิสต์ใหม่ขึ้นมาจากสมาชิกใน a ที่ไม่อยู่ใน b
4. จงเขียนเมธอด findPrevious(Object x) และ find(Object x) ของลิงค์ลิสต์โดยใช้อิเทอเรเตอร์ในการลูป
5. จงเขียนเมธอด void tail() ของลิงค์ลิสต์ โดยเมธอดนี้จะเอาสมาชิกตัวแรกของลิงค์ลิสต์ทิ้งไป
6. จงเขียนคิวขึ้นมาใหม่โดยใช้ลิงค์ลิสต์ แต่คราวนี้ให้ใส่สมาชิกใหม่ที่หัวคิว และเอาสมาชิกออกจากท้ายคิวแทน
7. จงเปลี่ยนโค้ดของสแตกและคิวที่สร้างจากอาร์เรย์ให้สามารถรับสมาชิกเป็นจำนวนเท่าไรก็ได้
8. จงวาดขั้นตอนในการใช้สแตกทำ (x+y)*z ให้เป็นพรีฟิกส์และโพสต์ฟิกส์
9. จงวาดขั้นตอนในการใช้สแตกทำ a+b-c * (d-e) ให้เป็นพรีฟิกส์และโพสต์ฟิกส์
10. จงวาดขั้นตอนในการใช้สแตกทำเลข a +b*c + (d*e+f) * g ให้เป็นพรีฟิกส์
11. จงวาดขั้นตอนในการใช้สแตกทำ –b+sqrt(b*x – d*a*c)/(e*a) ให้เป็นโพสต์ฟิกส์
12. จงวาดสแตกเพื่อแก้โจทย์ 1 2 5 * 4 2 + + * 3 -
13. สมมติว่าเราเอา 1,2,3,4,5 ใส่สแตกตามลำดับ จากนั้นป๊อปของจากสแตกสี่ครั้ง สิ่งที่ป๊อปออกมาจากสแตกก็เอาไปใส่คิวที่ว่างอยู่ ถามว่าตัวเลขไหนจะถูกเอาออกจากคิวได้เป็นตัวที่สอง
14. จงเขียนคิวจากอาร์เรย์โดยไม่ให้มีตัวแปร size
15. จงเขียนเมธอดที่ทำการหาแฟกทอเรียลของจำนวนเต็ม n โดยห้ามใช้ลูป จากนั้นจงวาดการเปลี่ยนแปลงของสแตกเฟรมที่เกิดจากการเรียกใช้ factorial(3)
16. จงเปรียบเทียบเวลาในการทำงาน ระหว่างสแตกที่สร้างจากอาร์เรย์ กับสแตกที่สร้างจากลิงค์ลิสต์
17. จงเปรียบเทียบเวลาในการทำงาน ระหว่างคิวที่สร้างจากอาร์เรย์ กับคิวที่สร้างจากลิงค์ลิสต์

18. จงเขียนโปรแกรม int power(int x, int y) หาค่าของ x ยกกำลัง y โดยใช้สแตก

19. จงเขียนโปรแกรม boolean isPalindrome(String s) หาว่า s อ่านจากหลังมาหน้าได้เหมือนกับอ่านจากหน้าไปหลังหรือไม่ โดยใช้สแตก

20. จงออกแบบสร้างตารางสปาร์ซโดยใช้อาร์เรย์ทั้งหมด แล้วจงบอกวิธีการบวกเลขจากตารางสปาร์ซสองตารางเข้าด้วยกัน

21. มีโค้ดการคูณจำนวนเต็มบวกดังนี้

```
1:    int multiply(int x,int y){
2:     int count = y;
3:     if (y= =0){
4:        return 0;
5:     } else {
6:        return (x+ multiply(x,y-1));
7:     }
8:    }
```

จงเขียนโค้ดนี้ใหม่ ให้เป็น tail recursion

22. ถ้ามีฟังก์ชั่นแฟคทอเรียลแบบ tail-recursive

fact(a,n) = a , if n=0

= fact(a*n,n-1) , otherwise

เราสามารถเรียกใช้ได้ด้วยการเรียก fact(1,n)

เราเอาฟังก์ชั่นนี้เอามาเขียนเป็นลูปได้ดังนี้

```
1:    int fact(int n){
2:    int a =1;
3:     while(n!=0){
4:        a=a*n;
5:        n=n-1;
6:    }
7:    return a;
8:    }
```

จงเอาฟังก์ชั่น tail-recursive ในรูปแบบข้างล่างนี้ไปเขียนใหม่ให้เป็นลูป

f(x) = a1, if c1

= a2, if c2

= a3, if c3

= f(x1), if d1

= f(x2), if d2

= f(x3), otherwise

23. ถ้ามีเขาวงกตดังเช่นในรูป

1 1 1 1 1 1

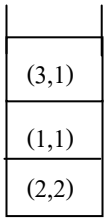
1 1 1 0 0 1

1 0 0 0 F 1

1 0 0 S 1 1

1 1 1 1 1 1

โดย 1 คือกำแพงและ 0 คือทางเดิน ให้ S เป็นจุดเริ่มต้นและ F เป็นจุดสุดท้าย (ทั้ง 2 จุดเป็นส่วนของทางเดิน) เราสามารถหาทางจาก S ไป F โดย เก็บค่า (X, Y) ที่ไปได้จากตำแหน่งปัจจุบันลง stack (บน ล่าง ซ้าย ขวา ตามลำดับ) เมื่อให้ค่าพิกัดของเลข 1 ซ้ายล่างสุดเป็น (0, 0) ฉะนั้น ตอนแรก stack จะเป็น



การเดิน เราจะ pop stack แล้วเดินไปจุดที่ pop (สมมติว่ายังไงก็เดินได้) แล้ว push ข้อมูลของที่ว่างรอบจุดที่พึ่งจะเดินไปถึงลง stack อีก (ไม่เอาจุดที่เคยเดินผ่านมาแล้ว) ทำซ้ำเรื่อยๆ จนเจอจุดหมาย เมื่อเจอแล้วไม่ต้อง push อะไรลง stack อีก ถามว่า ค่าสุดท้ายบน stack จะเป็นเท่าไร

24. ถ้าให้ L เป็นลิงค์ลิสต์ใดๆ และ P เป็นลิงค์ลิสต์ที่มีสมาชิกเป็นจำนวนเต็มที่จัดเรียงจากน้อยไปมาก จงเขียนเมธอด LinkedList specificElements(L, P) ซึ่งจะสร้างลิงค์ลิสต์ใหม่จาก L โดยดึงสมาชิกมาตามที่บอกไว้ใน P นั่นคือ ถ้า P มีสมาชิกคือ 1,3,4,6 ลิงค์ลิสต์ตัวใหม่จะเกิดจากการเอาสมาชิกตัวที่ 1,3,4 และ 6 ของ L มาสร้าง

25. ถ้าเรามีลิงค์ลิสต์อยู่ตัวหนึ่ง เราจะสลับสมาชิกสองตัวที่อยู่ติดกันในลิสต์ได้อย่างไร โดยใช้การเปลี่ยนลิงค์เฉยๆ

26. ถ้าเราต้องการเอาส่วนของลิงค์ลิสต์ตั้งแต่สมาชิกที่ถูกชี้ด้วยอิเทอเรเตอร์ p1 ไปจนถึงสมาชิกที่ถูกชี้ด้วยอิเทอเรเตอร์ p2 ไปใส่ไว้ข้างหน้าสมาชิกที่ถูกชี้ด้วยอิเทอเรเตอร์ p3 เราจะต้องเขียนโค้ดอย่างไร สมมติให้อิเทอเรเตอร์ทุกตัวนั้นถูกต้อง และช่วงของ p1 ถึง p2 ครอบคลุมอย่างน้อยหนึ่งสมาชิก

27. ถ้าเราเขียนสถานการณ์จำลองของร้านขายของเพื่อหาเวลารอเฉลี่ยโดยไม่ใช้คิวแต่ใช้ลิงค์ลิสต์แทน เราอาจไม่จำเป็นต้องสร้างคิวของลูกค้าขึ้นมาเลยก็ได้ ถามว่าจะต้องทำอย่างไร

28. จงเขียนเมธอด insertBefore(Object x, LinkedListItr p) ของลิงค์ลิสต์เพื่อเอา x ใส่ไว้ข้างหน้าของสมาชิกที่ถูกชี้ด้วย p

29. จงเขียนโค้ดของลิงค์ลิสต์แบบสองทาง ให้มีโนดปลอมอยู่ท้ายลิสต์และมีเมธอดที่รีเทิร์นอิเทอเรเตอร์ที่ชี้ไปที่โนดที่เก็บสมาชิกตัวสุดท้ายในลิสต์

30. จงเขียนเมธอด insertBefore(Object x, LinkedListItr p) และ insertAfter(Object x, LinkedListItr p) เพื่อใช้กับลิงค์ลิสต์แบบสองทางโดยเฉพาะ

31. จงเขียนเมธอด removeBefore(Object x, LinkedListItr p) และ removeAfter(Object x, LinkedListItr p) ของลิงค์ลิสต์ และลิงค์ลิสต์แบบสองทาง

32. ถ้าเราเก็บโพลีโนเมียลด้วยลิงค์ลิสต์ จงเขียนเมธอดในการบวกโพลีโนเมียลที่เก็บด้วยวิธีนี้

33. จงเขียนโค้ดของเมธอดในการคูณโพลีโนเมียลที่เก็บด้วยลิงค์ลิสต์ โดยหาวิธีทำให้ ค่า big O เป็น $O(M^2N^2)$, $O(M^2N)$ และ $O(MN\log(MN))$

34. จงเขียนโค้ดของเมธอด public void reverse() เพื่อเรียงสมาชิกภายในของลิงค์ลิสต์เสียใหม่โดยเรียงกลับกับของเดิม ให้ใช้เวลา O(N)

35. เกมส่งเหรียญเริ่มโดยมีคนอยู่ n คนนั่งล้อมวงเป็นวงกลม คนแรกมีเหรียญอยู่ในมือ เหรียญถูกส่งต่อไป m ครั้ง คนที่ถือเหรียญในครั้งที่ m จะต้องออกจากเกมไป แล้วการส่งต่อเหรียญจะเริ่มใหม่นับจากคนที่อยู่ถัดจากคนที่ออกจากเกมนั้น การส่งเหรียญจะมีไปเรื่อยๆจนเหลือคนสุดท้ายอยู่คนเดียวที่ไม่ออกจากเกม คนนั้นถือว่าเป็นผู้ชนะ จงเขียนโปรแกรมหาผู้ชนะเมื่อมีค่าอินพุตเป็น m และ n

36. จงเขียนโค้ดของลิสต์ที่จัดตัวเองได้ ใช้การจัดตัวทั้งสี่แบบ